Fast, Sound and Precise Narrowing of the Exponential Function

Timothy Hickey*and Qun Ju

Michtom School of Computer Science
Volen Center for Complex Systems
Brandeis University, Waltham MA, 02254

March 27, 1996

Abstract

In this paper we present an algorithm for narrowing the constraint $y = e^x$. The algorithm has been designed to be fast by using only IEEE multiplication. The main difficulty is to design algorithms which soundly, rapidly, and precisely compute upper and lower bounds on e^x and $\ln(y)$. We prove that our algorithms are correct and produce upper and lower bounds which differ by at most 2 ULP. Experiments indicate that the bounds differ by 1 ULP 99.8% of the time. The method we describe is table-driven as is Gal's accurate tables method, but the table is easier to construct and hence this method may be applicable to wider classes of functions.

1 Introduction

One of the most exciting advances in constraint logic programming over the last few years has been the incorporation of the relational interval arithmetic constraint solving algorithm in CLP systems (see CLP(BNR) [16] and CLP(F) [11]). Relational interval arithmetic is an extensions grows out of earlier work by Moore [15] on interval arithmetic. These new CLP languages allow the user to solve complex arithmetic constraints involving not just the algebraic functions (+,-,*,/), but a wide range of other functions as well [2].

One of the distinguishing features of CLP(intervals) interpreters is that they are sound, i.e., if S is any set of constraints on a set V of variables, and if the interpreter narrows the ranges of the variables in V, then all solutions to the

*This research was partially supported by NSF grant CCR-9403427

system S are guaranteed to be contained in the narrowed intervals. In other words, CLP(intervals) interpreters use algorithms which reduce interval widths and never omit any solutions.

To prove the soundness of a CLP constraint solver which is based on interval arithmetic, it is necessary and sufficient to prove that the narrowing algorithms associated to each primitive constraint are sound. For the narrowing of addition and multiplication, this is a relatively straightforward problem, since the IEEE 754 standard [12] for double precision arithmetic allows the user to specify the rounding mode for the primary operations (+,-,*,/) and the square root. In this paper we show how to implement a sound narrowing algorithm for constraint $y = x^x$.

From a practical point of view, the narrowing operators should be not only sound but also fast, since they are the dominant operation in an interval arithmetic constraint solving system. They should also be precise, that is, the upper and lower bounds returned by the narrowing procedure for a relation should be as close as possible to the mathematically tightestest upper and lower bounds. Finally, the space taken by the narrowing procedure should be relatively small. There is a natural tradeoff between speed, space usage, and precision of the narrowing procedures.

We begin with a quick description of the IEEE 754 standard for floating point arithmetic, which motivates our more general "floating point" domain. We then prove some Propositions concerning the result of evaluating certain types of expressions with various rounding modes selected.

Finally, we describe how to narrow the relation $y = e^x$, rapidly, soundly, and precisely using only floating point arithmetic operations. The narrowing algorithm for e^x relies on an algorithm for computing the rapidly calculate precise upper and lower floating point bounds on e^x and $\ln(x)$ for floating point numbers x. For the exp (resp. ln) function we prove soundness of the upper and lower bound algorithms, and we also prove that the computed bounds for e^x usually differ by one unit in the last place (ULP) and at worst differ by two ULPs. These results are supported by experimental evidence.

The method that we use is a variation of Gal's accurate tables approach [9], but our tables are somewhat simpler and more generally applicable. Just as with Gal's method, there is a simple test for "dangerous" situations which might result in 2 ULP errors and in this case, one can apply Ziv's observation [17] and call a multiprecision routine (possibly many times with increasingly larger precision) to attain the best possible upper (or lower) bound.

Finally, we discuss our plans to use the techniques described in this paper to implement narrowing routines for other useful special functions.

2 Preliminaries

2.1 IEEE Arithmetic and Sound Interval Arithmetic

Floating-point arithmetic is ubiquitous in scientific computing, but as is well-known, most floating-point operations produce mathematically incorrect results due to the limited number of bits allowed in the representation of real numbers. Luckily, most modern computers rely on the IEEE 754 standard to determine how these unavoidable roundoff errors should be handled. Thus, although all machines produce mathematically incorrect results, if they use the IEEE standard, then they produce the same incorrect results, at least for the basic arithmetic operations (+.-.*,/). The standard does not however address the roundoff error of the special functions (exp,log,sin,cos,sinh, bessel functions, etc.). Our goal in the following section is to show how to compute upper and lower bounds for these functions with a provably small error using the arithmetic operations as described by the IEEE standard. A good introduction to floating point arithmetic is Goldberg's survey [10].

2.2 Overview of the IEEE 754 standard

IEEE 754 is a standard for binary floating-point arithmetic, which specifies the precise layout in different precisions such as single and double precisions. According to IEEE 754 standard, which for simplicity we shall call the IEEE standard through out this paper, Numbers in the binary floating-point formats are composed of 3 fields:

1. (Sign Field) A 1-bit sign s.
2. (Exponent Field) A biased exponent $e_1e_2\cdots e_m$, where $m = 11$ for double precision ($m = 8$ for single precision). We define $E = e_1e_2\cdots e_m - bias$, where $bias = 2^{m-1}-1$, It is clear that $E_{min} = 0-bias = -bias = 2^{m-1}-1$, $E_{max} = 2^m - 1 - bias = 2^{m-1}$.
3. (Fraction Field) A fraction $x_1x_2\cdots x_n$. where $n = 52$ for double precision ($n = 23$ for single precision). Define $f = 0.x_1x_2\cdots x_n$.

The value of the number X represented by a single or double precision bit string is determined as follows:

1. (normal case) when $E_{min} < E < E_{max}$: $X = (-1)^s 2^E 1.x_1x_2\cdots x_n$
2. (denormalized case) when $E = E_{min}$: $X = 0$ if if $x_1x_2\cdots x_m = 00\cdots 0$, else $X = (-1)^s 2^E 0.x_1x_2\cdots x_n$
3. (NaN and ∞ case) when $E = E_{max}$: $X = \infty$ if $x_1x_2\cdots x_m = 00\cdots 0$, else X is NaN

Observe that two different binary bit strings represent different floating-point number(including $+\infty$, $-\infty$) except for the case of NaN and zero. There are many different data formats for the NaN. There are exactly two representations for number 0 because s can be either 0 or 1, we call it $+0$, -0 respectively. $+0$ and -0 are the same in all arithmetic operation, except one case $1/0$. Strictly saying, if the division-zero exception (see next paragraph) is disable (which exactly is the default situation), $1/+0$ will return $+\infty$, $1/-0$ willreturn $-\infty$.

The IEEE standard requires that the exceptions for invalid operation, division-by-zero, overflow, underflow, and inexact should be capable of being enabled/disabled separately or in groups.

Also there are 4 rounding modes, i.e, round toward nearest(RN), zero(RZ), $+\infty$(RP), and $-\infty(RM)$. The rounding operation is specified by IEEE standard that rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in destination's format while signaling the inexact exception. Except for binary-decimal conversion, operation such as add, subtract, multiply,divide, square root, round to integer, shall performed as if it first produced an intermediate result correct infinite precision and with unbounded range, and then round that result according to the current rounding mode. It is obvious that the operations, such as floating-point compare, absolute value, negative operation, are exact.

2.3 General Floating Point Numbers

In this section we describe the classes of floating point numbers considered here. We use this class so that the results may be extended to hardware that supports larger floating point numbers that the standard double precision (e.g. 96 bit, 128 bit, or more).

The algorithms described in this paper are designed to compute upper and lower bounds on the elementary functions while using only the generalized floating point arithmetic described in this section.

The set of floating point numbers will be denoted $\mathcal{R} = \mathcal{R}(emax, emin, F)$ where F is the number of bits used to represent the fractional part of the number and $[emin, emax]$ is the range of base 2 exponents allowed in this representation. If we let E denote the number of bits needed to represent the exponent, then $E =$ ceiling$(\log_2(emax - emin))$. A floating point number x can then be represented by three bit strings s (the sign bit), e (the exponent bits), and f (the fraction bits) of lengths 1, E, and F and the representation is the obvious generalization of IEEE 754, where we simply change the size of the exponent and fraction fields of IEEE 754 from (11,52) to (E,F). In this new context, we use the same definitions of denormalized numbers, biased exponents, positive and negative infinity, and NANs.

We let $\mathbf{R}$ denote the set of real numbers and we let ϕ^- (resp. ϕ^+) denote the "rounding mode map" which maps $\mathbf{R}$ to $\mathcal{R}$ by rounding down (resp. up) to the nearest element of $\mathcal{R}$:

$$\phi^- : \mathbf{R} \to \mathcal{R} \quad \phi^-(x) = \sup(r \in \mathcal{R}, r \leq x)$$
$$\phi^+ : \mathbf{R} \to \mathcal{R} \quad \phi^-(x) = \inf(r \in \mathcal{R}, x \leq r)$$

We let $\oplus, \otimes, \ominus, \oslash$ denote the floating point implementations of the standard arithmetic operators $+, *, -, /$. When the rounding mode has been set to round toward positive infinity, we indicate this by placing a bar over the operator, as with $\overline{\oplus}, \bar{\otimes}, \bar{\oslash}, \bar{\ominus}$. Similarly, an underbar is used to denote rounding toward negative infinity. Thus,

$$a \underline{\oplus} b = \phi^-(a+b) \;\; a \underline{\ominus} b = \phi^-(a-b) \;\; a \underline{\otimes} b = \phi^-(a*b) \;\; a \underline{\oslash} b = \phi^-(a/b)$$
$$a \overline{\oplus} b = \phi^+(a+b) \;\; a \bar{\ominus} b = \phi^+(a-b) \;\; a \bar{\otimes} b = \phi^+(a*b) \;\; a \bar{\oslash} b = \phi^+(a/b)$$

Recall also that $\mathcal{R}$ is the union of a finite set of real numbers with the non-numbers $\{+\infty, -\infty, nan\}$. We will let σ denote the successor function of $\mathcal{R}$:

$$\sigma(x) = inf(\{r \in \mathcal{R} : x < r\}) \text{ if } x \in \mathcal{R} - \{+\infty, nan\}$$

and for the non-numbers we define $\sigma(+\infty) = +\infty$, $\sigma(nan) = nan$. We also define $\mu(x)$ to be a unit in the last place (ULP) of a number $x \in \mathcal{R}$, that is:

$$\mu(x) = \sigma(|x|) - |x|$$

This is only defined for numbers x such that $\sigma(|x|) \notin \{+\infty, nan\}$. When we say that a and b are k ULP apart, we will mean precisely that

$$\sigma^k(a) = b$$

That is, we can get from a to b by successively applying the operator which adds one unit in the last place:

$$x \mapsto x + mu(x)$$

Observe that the concept of 1 ULP depends on base 2 exponent of the floating point whose unit in the last place is being considered. If x and y are to floating points with the same base 2 exponent, i.e., $\lambda(x) = \lambda(y)$, then they will have the same ULP. We will define $\mu(\beta)$ for any real number β by

$$\mu(\beta) = \mu(\phi^-(|\beta|))$$

Thus, if β lies between two floating point numbers, x and y, then $\mu(\beta)$ will be the minimum of $\mu(x)$ and $\mu(y)$.

2.4 Some properties of arithmetic using IEEE rounding modes

Note that setting the rounding mode to round toward minus infinity does not guarantee that every assignment $v = E$ will produce a lower bound for the expression E. For example, if $x = -x0$ is negative and y is positive, then the assignment $v = x * (1 + y)$ will proceed by evaluating $1 + y$ and rounding toward minus infinity to get $1 + y - e1$. It will then multiply this by x and round toward minus infinity again to get:

$$x \underline{\otimes} (1 \underline{\oplus} y) = x * (1 + y - e1) - e2 = -(x0 * (1 + y)) + x0 * e1 - e2$$

Thus, depending on the relative sizes of the errors $e1, e2$ and x, the result stored in v could be either above, below, or equal to the true value of E. There are however two special cases where the result of evaluating an expression E with the rounding set to RM (resp. RP) has the same effect as evaluating the expression in "infinite precision" arithmetic and then rounding toward minus infinity (resp. plus infinity). We describe these two useful cases in the following two propositions.

Proposition 1 *Let $a_1, \ldots, a_n \in \mathcal{R}$ be floating point numbers such that for each i, $a_i \geq (a_{i+1} + \ldots + a_n) \geq 0$. Then,*

$$a_1 \underline{\oplus} (a_2 \underline{\oplus} \ldots (a_{n-1} \underline{\oplus} a_n) \ldots) = \phi^-(a_1 + \ldots + a_n)$$
$$a_1 \overline{\oplus} (a_2 \overline{\oplus} \ldots (a_{n-1} \overline{\oplus} a_n) \ldots) = \phi^+(a_1 + \ldots + a_n)$$

Proposition 2 *Let $a_1, \ldots, a_n \in \mathcal{R}$ be floating point numbers such that for each i, $|a_i| \geq 2|a_{i+1} + \ldots + a_n|$. Then,*

$$a_1 \underline{\oplus} (a_2 \underline{\oplus} \ldots (a_{n-1} \underline{\oplus} a_n) \ldots) = \phi^-(a_1 + \ldots + a_n)$$
$$a_1 \overline{\oplus} (a_2 \overline{\oplus} \ldots (a_{n-1} \overline{\oplus} a_n) \ldots) = \phi^+(a_1 + \ldots + a_n)$$

These two propositions allow us to conclude that by properly setting the rounding mode to round toward positive (resp. negative infinity) the result of evaluating certain decreasing sums of floating point numbers will be an upper bound (resp. lower bound) of the sum which is precise to the last bit.

Corollary 1 *Let $a_1, \ldots, a_n \in \mathcal{R}$ be floating point numbers as in either of the two previous propositions, and let*

$$v_- = a_1 \underline{\oplus} (a_2 \underline{\oplus} \ldots (a_{n-1} \underline{\oplus} a_n) \ldots) v_+ = a_1 \overline{\oplus} (a_2 \overline{\oplus} \ldots (a_{n-1} \overline{\oplus} a_n) \ldots)$$

Then, $v_- < v_+$ are consecutive floating point numbers, i.e. $\sigma(v_-) = v_+$.

The two propositions above follow easily from the following lemma by induction on n. The $n = 2$ case is trivially true. Assuming now the propositions are true for $n - 1$, we get the induction case from the lemma by setting

$$\beta = a_2 + a_3 + \ldots + a_n$$

and $a = a_1$.

Lemma 1 *Let $a \in \mathcal{R}$ be any floating point number and let $\beta \in \mathbf{R}$ be any real number such that either $a \geq \beta \geq 0$ or if $|a| > 2|\beta|$, then*

$$\phi^-(a+\beta) = \phi^-(a+\phi^-(\beta))$$
$$\phi^+(a+\beta) = \phi^+(a+\phi^+(\beta))$$

Proof. Let us first prove the following claim:

Claim 1 *that if $s \in \{+,-\}$ then*

$$\phi^s(a+\beta) = \phi^s(a+\phi^s(\beta))$$

is true if

$$\phi^s(a+\beta) - a \in \mathcal{R}$$

In the case where $s = +$, we have

$$c = \phi^+(a+\beta) \leq \phi^+(a+\phi^+(\beta)) = d$$

If the lemma is false then we must have $c < d$. Thus, from the definition of ϕ^+ we get

$$a + \beta \leq c < a + \phi^+(\beta) \leq d$$

And if we subtract a from both sides we get

$$\beta \leq c - a < \phi^+(\beta) \leq d$$

Observe now that if $c - a \in \mathcal{R}$, then by the definition of ϕ^+ we must have

$$\phi^+(\beta) = c - a$$

which is a contradiction. The case where $s = -$ can be proved in the same way. We can also reduce to the previous case using the fact that

$$\phi^-(a) = -\phi^+(-a) \quad c - a \in \mathcal{R} \Leftrightarrow -(c-a) \in \mathcal{R}$$

To conclude the proof of the lemma, we will show the following:

Claim 2 *Let $a \in \mathcal{R}$ and let $\beta \in \mathbf{R}$, and let $s \in \{+,-\}$, then*

$$\phi^s(a+\beta) - a \in \mathcal{R}$$

is true if

- $a \geq \beta \geq 0$, *or*
- $|a| \geq 2|\beta|$

Consider first the case where $a \geq \beta \geq 0$. If $a = \beta$ then $a+\beta = 2a \in \mathcal{R}$ and the lemma is trivially true, so we may assume that $a > \beta$. In this case, let $\gamma = a+\beta$ and let $c = \phi^s(a+\beta)$ and $\alpha = c-a$. Observe that $2a \geq c \geq a \geq c-a \geq 0$. Thus, if $\lambda(c)$ denotes the base two exponent of c, then $\lambda(c) \geq \lambda(a) \geq \lambda(c-a)$. Thus, if we let $\mu(a)$ be the power of two which denotes one unit in the last place of a, then $\mu(c) \geq \mu(a) \geq \mu(c-a)$. But $c-a$ is formed from bit-by-bit subtraction and so $\mu(c-a) = \mu(a)$. Since $\lambda(c-a) \leq \lambda(a)$, we see that $c-a$ can be expressed with a fractional part of at most F bits and since the last non-zero bit in $c-a$ is in no smaller a position than $\mu(a)$, this holds even if $\mu(c-a) < emin$ and $c-a$ is denormalized.

Consider now the case where $|a| \geq 2|\beta|$. We may assume without loss of generality, that $a > 0$ and $\beta < 0$. Again letting $c = \phi^s(a+\beta)$, we find that

$$a \geq c \geq a/2 \geq a-c \geq 0$$

Thus, we see that $\lambda(c) \in \lambda(a)+[0,-1]$ and $\lambda(a-c) \leq \lambda(a)-1$. Since $\mu(a-c) \geq \mu(c)$ we must be able to represent $a-c$ with a fractional part of at most F bits and with the position of its last nonzero bit no smaller than $\mu(c)$. Thus, $a-c \in \mathcal{R}$. $\square$

3 Sound Narrowing for exp and log

In this section we describe a fast, sound and precise implementation of the the procedure:

```
BOOLEAN narrow_exp(INTERVAL *X,INTERVAL *Y);
```

which accepts two intervals X and Y whose endpoints are floating point numbers (possibly infinite).

The `narrow_exp` procedure narrows the intervals X and Y in such a way that if $x \in X$ and $y \in Y$ before the call and if $e^x = y$, then x and y will still belong to the narrowed intervals after the call. If the intervals are narrowed to the empty interval then `narrow_exp` returns `FALSE` otherwise it returns `TRUE`. Since exp and log are inverse functions we can implement `narrow_log(X,Y)` as a call to `narrow_exp(Y,X)`. Moreover, since they are both monotone increasing functions the narrowing procedure is particularly simple and is shown in Figure 1. The procedure

```
void intersect(INTERVAL *X,double lo,double hi,BOOLEAN *v)
```

```
boolean narrow_exp(interval *x, interval *Y){
 boolean val=true;
 intersect(Y,exp_clp(lo,X->lo),exp_clp(hi,X->hi),&val);
 intersect(X,log_clp(lo,Y->lo),log_clp(hi,Y->hi),&val);
 return(val);
}

void intersect(interval *X, double vlo, double vhi, boolean *val)
{
  if (!(*val)) return;
  if (vhi < X->hi) X->hi = vhi;
  if (X->lo < vlo) X->lo = vlo;
  *val = (X->lo <= X->hi);
}
```

Figure 1: Procedure `narrow_exp`

intersects the interval `*X` with the interval $[lo, hi]$ and stores the value back in `X` provided the result is non-empty and `*v = TRUE`, otherwise it returns with `*v = FALSE`. The two functions

```
double exp_clp(int lohi,double x);
double log_clp(int lohi,double x);
```

compute upper and lower bounds on e^x and $\ln(x)$, that is, for all x we have:

```
  exp_clp(lo,x) <= exp(x) <= exp_clp(hi,x)
  log_clp(lo,x) <= ln(x)  <= log_clp(hi,x)
```

Assuming the correctness of these auxiliary procedures it is clear that `narrow_exp` is sound and that if the `exp_clp` and `log_clp` procedures were exact, then `narrow_exp(X,Y)` would narrow X and Y to their theoretically minimal sizes.

In the next two subsections we will show how to compute upper and lower bounds on the values of the exp and ln functions quickly, soundly. We will implement these functions using tables of precomputed values to quickly reduce to the evaluation of a rapidly converging powerseries.

3.1 Computing bounds on the exponential function

In this section we present and analyze a procedure for computing upper and lower bounds on e^x using a table based algorithm.

The procedure `exp_clp(lohi,x)` (shown in Figure 2) computes, for every floating point number $x \in \mathcal{R}$, an upper or lower bound ($v_+ \in \mathcal{R}$ or $v_- \in$

1. `tablesize=` 2^M is the size of the table used in the range reduction, and `expdeg=D` is the number of terms of the Taylor series used to approximate e^x for x on the reduced range.

2. `ea=exp_a[hilo][i]` and `eb=exp_b[hilo][i]`. These arrays store precomputed double precision numbers representing $z = e^{i/2^M}$ rounded to an accuracy of $2F+1$ significant bits, where ea represents the first $1+F$ bits and eb is the next $1+F$ bits. (Recall that a non-denormalized floating point number corresponds to a binary decimal of $F+1$ bits, including the initial 1.) Formally, they are defined as follows: let $s \in \{+1, -1\}$ denote the rounding mode and let $t = \phi^{\nu}_{2F+1}(e^{i/2^M})$, then $ea = \phi^{-}_{F}(t) > 0$, $eb = t - ea \geq 0$ and $eb \in [0, 2^{-F}]$, $e^{i/2^m} \in ea + eb + s * [0, 2^{-2F}]$.

3. `ln2_a[lohi]` and `ln2_b[lohi]` These constants store precomputed double precision numbers representing $\ln(2)$ rounded to an accuracy of $C+F$ significant bits, where $ln2a$ represents the first C bits and $ln2b$ is the next F bits. Formally, they are defined as follows: let $s \in \{+1, -1\}$ denote the rounding mode and let $t = \phi^{\nu}_{C+F+1}(\ln(2))$, then $ln2a = \phi^{-}_{C}(t) > 0$, $ln2b = t - ln2a \geq 0$ and $ln2b \in [0, 2^{-C}]$, $\ln(2) \in ln2a + ln2b + s * [0, 2^{-(C+F+1)}]$. *This definition depends on the paramter C.*

4. `ln2_plus_eps, ln2_minus_eps` These constants are defined by $\phi^{s}_{B}(\ln(2)) + s2^{-B-1}$ where s is the rounding mode (1 for RP, -1 for RM), i.e. round up (or down) to obtain a $B+1$ bit number and then add (or subtract) one unit in the last place. *This definition depends on the paramter B.*

5. `log_max_real`, `log_min_real`. These are constants $x, y \in \mathcal{R}$ with the property that $e^x > maxreal$ and $e^y < minreal$ where $maxreal$ is the largest finite number representable in $\mathcal{R}$, and $minreal$ is the smallest positive number representable in $\mathcal{R}$.

6. `max_pos[hilo], min_pos[hilo]`. These return the upper or lower bounds on e^x when x is too large or too small to be represented in $\mathcal{R}$.

```
max_pos[hi] = pos_inf     min_pos[hi] = min_real
max_pos[lo] = max_real    min_pos[lo] = 0
```

7. `two_to_n(hilo,n)`. This returns 2^n rounded either up or down (depending on `hilo`) to the nearest element of $\mathcal{R}$:

```
two_to_n(s,n) = max_pos[s]  if emax <= n
              = 2^n         if emin-F < n < emax
              = min_pos[s]  if n <= emin-F
```

Table 1: Precomputed constants used in `exp_clp`

```
double exp_clp(int hilo,double x) {
  double a,ea,eb,z,zea,zeb,v,y,a1,ln2_est,aa;
  int i,n,j,lohi,hilo2,d;
/*1*/
       if(x> ln_max_real) return(max_pos[hilo]);
       if(x< ln_min_real) return(min_pos[hilo]);
/*2*/
       fpsetround(round_mode[hilo]); lohi = 1-hilo;
       hilo2 = (x>0)?hilo:lohi;
/*3*/
 /*a*/ if ((-2.0/tablesize < x) && (x < 2.0/tablesize))
         {i=0; n=0; a=x;}
       else {
 /*b*/   ln2_est = (x>0)?ln2_plus_eps:ln2_minus_eps;
 /*c*/   n=floor(x/ln2_est);
 /*d*/   y = x + (n*(-ln2_a[hilo2]));
 /*e*/   i = (y*tablesize);
 /*f*/   if ((i>0)&&(n!=0)) i=i-1;
 /*g*/   a1=y-((double) i)/tablesize;
 /*h*/   a = a1 + (n*(-ln2_b[hilo2]));
       }
/*4*/
 /*a*/ aa = a*a; d = (expdeg/2)*2 + 1;
 /*b*/ z = ((hilo==LO)?0.5:2.0)*a/(d+2);
 /*c*/ for(j=d;j>=1;j-= 2)
 /*d*/    z = ((z+1)*aa/(j+1)+ a)/j;

/*5*/
       ea = expa[hilo][i];  eb = expb[hilo][i];
       zea = z*ea; zeb = z*eb;
       if (zea > eb)
         v= (ea + (zea + (eb + zeb)))*two_to_n(hilo,n);
       else
         v= (ea + (eb + (zea + zeb)))*two_to_n(hilo,n);
  return(v);
}
```

Figure 2: The `exp_clp` procedure.

$\mathcal{R}$), depending on the value of the parameter $lohi \in \{lo, hi\}$. We will prove the correctness of this procedure (relative to some assumptions about number system $\mathcal{R}$. We also provide experimental evidence (using IEEE double precision arithmetic) which shows that these bounds usually differ by one unit in the last place (ULP) and at worst differ by two ULPs. The probability that the error is two ULPs can be decreased is inversely proportional to the size of the table used. For a table of size 2^{10}, the experimentally observed probability of having a 1 ULP error is is greater than 99.8%.

The reason that the computed upper and lower bounds for e^x usually differ by at least one ULP is that for all rational numbers x (except 0), e^x is irrational. [1] Thus, e^x is not exactly equal to a floating point number, and so the upper and lower bounds cannot be equal and hence must differ by at least one ULP.

The reason that it is difficult to guarantee that the bounds always differ by at most one ULP has to do with the table maker's dilemma. Suppose that e^x is very close to a floating point number f, then in order to determine the floating point numbers on either side of e^x, we must decide on which side of f it lies, and it may take considerably more work to make this determination than to find f in the first place. Ironically, the cases in which we cannot compute the upper and lower bounds to within 1 ULP of each other are the times when e^x is very close to the single floating point number f between the two bounds, and hence these are the cases when we know the value of e^x most accurately.

3.1.1 The mathematical algorithm

If we temporarily ignore the problem of handling the roundoff error, the algorithm we use to compute $y = e^x$ using a table of size 2^m is quite standard. We simply make a range reduction using the addition law of exponentials as follows:

1 Let $n = floor(x/\ln(2))$, $u = x - n * \ln(2)$, $i = floor(i * 2^m)$, and $a = u - i/2^m$, so that

$$x = a + i/2^m + n * \ln(2) \quad e^x = e^a e^{i/2^m} 2^n$$

2 Compute an approximation z of e^a using a Taylor polynomial where R is an estimate for e^ξ for some $\xi \in [0, a]$:

$$z = \left(\sum_{i=0}^{d-1} a^i/i! \right) + Ra^N/N!$$

3 Lookup the value e_i of $e^{i/2^m}$ in a precomputed table, and use e_i, z, and n to compute e^x:

$$e^x = z * e_i * 2^n$$

[1] Indeed, this follows easily from the (rather subtle) property that e is a transcendental number, i.e., it satisfies no polynomial with rational coefficients.

$$
\begin{aligned}
F > F - E \geq C \geq B + 3 > B \geq E + 4 > E \geq 11 \\
C - E - 1 \geq M \geq 3 \\
D \geq F/(M-1) + 2
\end{aligned}
$$

Table 2: Constraints on the parameters

The use of the table of precomputed values allows us to use a smaller number of terms in the Taylor series to attain the same degree of approximation.

3.1.2 Soundness of the program

The program in Figure 2 follows the same basic algorithm as described above, but requires some rather delicate programming to guarantee that the resulting value is an upper bound (or lower bound) depending on the value of the variable $lohi \in \{lo, hi\}$. The procedure makes use of several precomputed values which are defined and described in Table 1. These constants are defined in terms of the floating point system $\mathcal{R}(E, F)$.

In this section we provide a brief explanation of why the `exp_clp` procedure is sound and generates precise results. A complete proof is in the appendix. The main result is the following:

Proposition 3 *Let $x \in R$, and let y_B be the value computed by the call* `exp_clp(B,x)`*, where* `exp_clp` *is the procedure defined in Figure 2 using the constants defined in Table 1. Further assume that the integer parameters B, C, D, E, F, M appearing in the definitions of these constants satisfy the relations in Table 2. Then,*

$$y_{lo} < e^x < y_{hi}$$

Moreover, the distance between y_{hi} and y_{lo} is at most 2 ULP, and whenever the distance is exactly 2 ULP there is a unique floating point number y strictly between y_{hi} and y_{lo} and we must have

$$|e^x - y| < 2^{-M+3} ULP$$

Observe that the procedure is divided into 5 steps. We discuss the idea behind the code in each step in order.

Step 1. Here we handle the cases where e^x overflows or underflows

Step 2. Here we set the rounding mode to RP or RM depending on whether we are computing an upper bound or a lower bound.

Step 3. Here we find a floating point number a and two integers i and n such that:

$$x + s\epsilon_1 = a + i/2^M + n \ln(2)$$

s is the rounding mode (1 for RP and -1 for RM) and $\epsilon \in [0, 2^{-F-M+1}]$ The code is also designed to guarantee that $i \in [0, 2^M]$ and if $n \neq 0$ then $a \in [0, 2^{-M+1}]$ and if $n = 0$ then $a \in [-2^{-M+1}, 2^{-M+1}]$.

Step 4. Here we compute an approximation to z to $e^a - 1$ with a high degree of accuracy:

$$e^a = (1+z)(1-s\epsilon_2)$$

where $\epsilon_2 > 0$ and $\epsilon_2 \leq 2^{-M+2} ULP(1+z)$, i.e. $\epsilon_2 \in [0, 2^{-F-M+2}$

Step 5. At this point we combine the previous computations very carefully. More precisely, this code is designed so that the error between v and the sum $v' = ea + (zea + (eb + zeb))$ is at most 1 $ULP(v)$, i.e., for some $u \in [0,1)$ we have

$$v' = v(1 - su2^{-F})$$

Also the code has been designed so that the relative difference between v' and e^x is at most a small fraction of a ULP of v, i.e., for some $u' \in [0,1)$ we have:

$$\begin{aligned}
e^x &= e^a e^{i/2^M} 2^n e^{-s\epsilon_1} \\
&= (1+z)(ea+eb)((1-s\epsilon_2)(1-s\epsilon_3)e^{-s\epsilon_1} \\
&= (ea + z*ea + eb + z*eb)((1-s\epsilon_2)(1-s\epsilon_3)e^{-s\epsilon_1} \\
&= (ea + zea + eb + zeb)(1 - su'2^{-F-M+3}) \\
&= v'(1 - su'2^{-F-M+3}) \\
&= v(1 - su'2^{-F-M+3})(1 - su2^{-F})
\end{aligned}$$

Thus, the difference between e^x and `exp_clp(lohi,x)` is at most $1+\epsilon$ ULP for $\epsilon \leq 2^{-M+3}$. Since this is true for both the upper and lower bound, we see that the difference between the bounds is at most $2+2\epsilon$ ULP, and hence is at most 2 ULP. Moreover, the difference is 2 ULP only when e^x is within $2^{-M+3} ULP$ of a floating point number which we could expect to happen not more than 2^{-M+4} of the time.

3.1.3 Comparison with Gal's method

The main similarities between our method and Gal's are that we both use the standard range reduction method to reduce the argument to absolute value at most $\ln(2)$ and then use a table to reduce the argument to at most 2^{-M} where 2^M is the table size. At the innermost level, we use a Taylor series with remainder whereas Gal uses a minimax polynomial, but this isn't a crucial difference. The tables Gal uses to compute a function f are more complex than the ones presented here. He requires that the ith table entry T_i be the nearest floating point number to the value $v_i = f(x_i)$ of f at some point x_i near $i/2^M$. The point x_i must satisfy several constraints depending on the particular range reduction method being employed, but most importantly it must be such that

v_i should be within 2^{-M} ULPs of T_i. He provides a heuristic argument as to why one should always be able to find such an x_i, but does not provide a proof.

Our approach does not place any such restrictions on the table entries and in that sense is completely general. Rather than store x_i and T_i, we simply store the value at $i/2^M$ using two double precision numbers (the poor man's quadruple precision). We can get away with this simpler approach because our precision and soundness argument relies on some lemmas on the accuracy of certain sums evaluating using the upward (or downward) rounding modes.

3.1.4 Experimental results

We tested this procedure for various table sizes M (from 2 to 1024) on 100,000 random inputs in the range [-700,700]. The results are in table 3. Since we used IEEE 754 double precision arithmetic, we have $E = 11$ and $F = 52$ so the constraints on M,B,and C are:

$$\begin{aligned} F > F - E \geq C \geq B + 3 > B \geq E + 4 > E \geq 11 \\ 52 > 41 \geq C \geq B + 3 > B \geq 15 \\ C - E - 1 \geq M \geq 3 \\ C - 12 \geq M \geq 3 \\ D \geq 52/(M-1) + 2 \end{aligned}$$

In our experiments, we selected $C = 32$ and $B = 16$ and let M range in the interval $[1, 10]$.

In our experiments with $2^M \geq 256$, we found that the error was always less than 2 ULP even when Moreover, the fraction of cases where the error was 2 ULP was approximately $2^{-(M-3)} = 8/tablesize$.

3.2 Computing bounds on `log(x)`

The same ideas used to attain a fast, sound, and precise implementation of `exp_clp` can be used to compute `log_clp(lohi,x)`. We reduce the problem of computing $log(x)$ to the problem of computing $\log(1 + a)$ for $a < 2^{-m}$ by a range reduction which involves finding integers $i \in [0, 2^m]$ and n and a floating point number $a \in [0, 2^{-m}]$ such that

$$x(1 + s\epsilon_1) = (1 + a)(1 + i/2^m)2^n$$

where ϵ_1 is much smaller than the machine epsilon. Hence,

$$\ln(x) + \ln(1 + s\epsilon) = \ln(1 + a) + \ln(1 + i/2^m) + n * \ln(2)$$

As in the case of the exponential function, we evaluate $\ln(1 + a)$ using a Taylor series and $\ln(1 + i/2^m)$ is evaluated using a table. One complication of the algorithm for the log is that when x is a small negative number, the range

M	Tablesize	D	ULP=1	ULP=2	ULP>=3	MaxULP
1	2	61	49.944\%	43.019\%	7.037\%	4
2	4	55	50.144\%	38.874\%	10.982\%	4
3	8	29	54.900\%	45.100\%	0	2
4	16	21	74.201\%	25.799\%	0	2
5	32	15	86.570\%	13.434\%	0	2
6	64	13	93.045\%	6.955\%	0	2
7	128	11	96.507\%	3.439\%	0	2
8	256	11	98.312\%	1.688\%	0	2
9	512	9	99.147\%	0.853\%	0	2
10	1024	9	99.565\%	0.435%	0	2

Table 3: Benchmarks for `exp_clp`

reduction method described above will select $n = -1$ and $i = 2^{-m} - 1$ and this results in a subtraction of two nearly equal terms $\ln(2)$ and $\ln(2 - 1/2^m)$. This can result in a considerably loss of accuracy. To circumvent this problem, we use a Taylor series whenever $|a|$ is sufficiently small. The program implementing this algorithm is shown in Figure 3. We ran an experiment, using a table size of 1024 and a Taylor degree of 10, and attained an error of 1 ULP 99.8% of the time and 2 ULP the remaining 0.2%, with no errors larger than 2 ULP.

4 Future Work

We are currently extending the methods described in this paper to the other special functions of scientific computing, including:

- The trigonometric functions. We have implemented sound narrowing for the relation $y = sin(x)$ using the methods described above and have obtained reasonably good initial results.
- The hyperbolic functions. This should be relatively easy, along the lines of the exponential and log functions.
- The error function, the bessel functions, gamma function, etc.

Acknowledgements

Also, we gratefully acknowledge the support of the W. M. Keck foundation.

```
double log_clp(int hilo,double x) {
  double a,z,v,aa,a0,f;
  int  n,i,j,lohi,hilo2;
/*1*/  if (x  < 0.0) return(nan);
       if (x == 0.0) return(-max_pos[lohi]);
       if (x == 1.0) return(0.0);
       if (x == pos_inf) return(max_pos[hilo]);
/*2*/
       fpsetround(round_mode[hilo]);
       lohi = 1-hilo;
/*3*/
       if ((x <= 1 + 4.0/tablesize)  && (x >= 1 - 4.0/tablesize) )
          {n=0; i=0; a=x-1;}
       else{
         compute_exponent_and_fraction(x,&n,&f);
         i = floor((f-1)*tablesize);
         if (i > 0) {i=i-1;}
         a0 = 1 + (double) i / tablesize;
         a = (f-a0)/a0;
       }
/*4*/
       aa = -((-a)*a);
       z=(hilo==LO)?0.0:1.0;  /* check this */
       for(j=2*logdeg;j>1;j -= 2)
         z = a/(j-1) + aa*(1.0/(-j) + z);
/*5*/
       hilo2 = (n<0) ? (1-hilo): hilo;
       v =
           (n*ln2_a[hilo2] + log_a[hilo][i])  /* exact addition */
            + (z +
           (n*ln2_b[hilo2] + log_b[hilo][i]));
       return(v);
}
```

Figure 3: The log_clp procedure.

References

[1] Bate, R. and Mueller, D. and White, J., Fundamentals of Astrodynamics. Dover, 1971.

[2] Benhamou, F. and Older, W., Applying interval arithmetic to real, integer, and boolean constraints. Journal of Logic Programming, (To Appear), 1994.

[3] Cohen, H. PARI reference manual 1996.

[4] Colmerauer, A., Solving Equations and Inequations on Finite and Infinite Trees. in Proceedings of the Conference on Fifth Generations Computer Systems, K.L. Clark and S.A. Tarnlund (eds.), 1984.

[5] Colmerauer, A., An Introduction to Prolog III. Communications of the ACM, july 1990.

[6] Eckmann, J-P. and Wittwer, P., Computer Methods and Borel Summability Applied to Feigenbaum's Equation. Springer-Verlag, Lecture Notes in Physics, 227, 1985.

[7] Computer Evaluation of Mathematical Functions Fike, C.T. Prentice Hall Series in Automatic Computation, Prentic-Hall Inc., 1968.

[8] Gal S., Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance, in Accurate Scientific Computations, (eds. Miranker, W.L., and Toupin, R.A.), Springer-Verlag Lecture Notes in Computer Science No. 235, Springer-Verlag, 1985.

[9] Gal, S. and Bachelis, B. , An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard, ACM Transactions on Mathematical Software, Vol. 17, No. 1, March 1991.

[10] Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic ACM Computing Surveys, Vol. 23, No. 1, 1991.

[11] Hickey, T. CLP(F) and Constrained ODEs ECRC Technical report Proceedings of the Workshop and Constraints and Modelling Ithica, NY, 1994

[12] IEEE standard 754-1985 for Binary Floating-Point Arithmetic IEEE. Reprinted in SIGPLAN 22(2), pp. 9-25, 1985

[13] Jaffar, J. and Lassez, J. L., Constraint Logic Programming. in Proceedings of the 14th ACM Symposium on the Principles of Programming Languages, 1987.

[14] Jaffar, J. and Michaylov, S., Methodology and Implementation of a Constraint Logic Programming System. in Proceedings of the Fourth International Conference on Logic Programming, MIT Press, 1987.

[15] Moore, R. E., Interval Analysis. Prentice-Hall, 1966.

[16] Older, W. and Vellino, A., Constraint Arithmetic on Real Intervals, in Constraint Logic Programming: Selected Research Colmerauer, A. and Benhamou, F. (eds), MIT Press 1993.

[17] Ziv, A., Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit, ACM Transactions on Mathematical Software, Vol. 17, No. 3, September 1991.

A The Soundness and Precision of `exp_clp`

In this section we prove that the `exp_clp` procedure in Figure 2 is sound and gives precise results with an error of at most 2 ULP.

Proposition 4 *Let $x \in R$, and for $B \in \{lo, hi\}$, let y_B be the value computed by the call* `exp_clp(B,x)`*, where* `exp_clp` *is the procedure defined in Figure 2 using the constants defined in Table 1. Further assume that the integer parameters B, C, D, E, F, M appearing in the definitions of these constants satisfy the relations in Table 2. Then,*

$$y_{lo} < e^x < y_{hi}$$

Moreover, the distance between y_{hi} and y_{lo} is at most 2 ULP, and whenever the distance is exactly 2 ULP there is a unique floating point number y strictly between y_{hi} and y_{lo} and we must have

$$|e^x - y| < 2^{-M+3} ULP$$

Proof. We will first prove this assuming that none of the operations result in overflow or underflow or denormalized numbers, i.e., we will assume that all operations performed in the algorithm result in numbers with exponents strictly between $emax$ and $emin$. We will examine the remaining cases at the end of the proof.

Step 1. We handle the overflow cases here. When $e^x > max\_real$ or $e^x < min\_real$, we can return the appropriate upper and lower bounds without further computation. We may now assume that

$$\begin{aligned} x &\in [log\_min\_real, log\_max\_real] \\ &\subset [log_2(min\_real), log_2(max\_real)] \\ &\subset [-2^{E-1}, 2^{E-1}] \end{aligned}$$

where we have used the fact that $log_2(x) > \ln(x)$ and that the base 2 exponents of min_real and max_real are within the stated bounds.

Step 2. Here we set the rounding mode which applies to all arithmetic operations in the rest of the procedure. When computing an upper bound for

e^x (i.e., when `hilo = HI`) we set the mode to round up. When computing lower bounds, we set the mode to round the result of each operation down. One of the important features of our system is that we use a single rounding mode throughout.

Let s be 1 (resp. -1) if `hilo = HI` (resp. `LO`), then we can express the result of an arithmetic operation under the current rounding mode as follows:

$$a \oplus b = (a + b) * (1 + s * v) \quad a \otimes b = (a * b) * (1 + s * v)$$
$$a \ominus b = (a - b) * (1 + s * v) \quad a \oslash b = (a/b)(1 + s * v)$$

where $v \in [0, 2^{-F}]$ provided the result is a finite number which is not denormalized.

Step 3. These lines of code are designed to find the decomposition $x + s * \epsilon = a + i/2^m + n * \ln(2)$ for some very small positive epsilon. We will now prove the following lemma.

Lemma 2 *At the end of Step 3, the integers n and i and the floating point numbers a and x have the following properties*

- *If $|x| < 2^{-M+1}$, then $a = x$ and $i = n = 0$,*
- *If $n = 0$ and $|x| \geq 2^{-M+1}$, then*

$$x = a + i/2^m$$
$$i \in [0, 2^M] \quad a \in [0, 2^{-M})$$

- *If $n \neq 0$, and $|x| \geq 2^{-M+1}$, then for some $\epsilon \in [0, 2^{-F-M+1}]$, we have*

$$x + s * \epsilon = a + i/2^M + n * \ln(2)$$
$$n \in [-2^E, 2^E] \quad i \in [0, (7/8)2^M] \quad a \in [0, (5/4)2^{-M+1})$$

Lets begin by examining **step 3a**. If this conditional is true, then we simply set $i = n = 0$ and $a = x$ and the lemma is true in this case. We can now assume that the condition is false and so $|x| \geq 2^{-M+1}$.

After **step 3b**, there is an ϵ_1 such that

$$\texttt{ln2\_est} = \ln(2) + s_x \epsilon_1 \quad \epsilon_1 \in [2^{-B-1}, 2^{-B}]$$

where s_x is the sign of x (i.e., 1 if $x \geq 0$ and -1 otherwise).

After **step 3c**, there is a real numbers $r, u \in [0, 1)$ such that

$$n + r = \frac{x}{\ln(2) + s_x \epsilon_1}(1 + su2^{-F})$$

and n is an integer. At this point, if $n = 0$ then we must have

$$x \in (\ln(2) + 2^{-B})/(1 - 2^{-F})[0, 1) \subset [0, 0.75)$$

where we have used the assumption that $F \geq B \geq E \geq 12$ to make a very rough estimate of the interval containing x. In the case where $n = 0$, we will let $y = x$ in step 3d and in steps 3e-3h, using the fact that $M < F$, we find an integer i and a floating point number a such that the following relation holds with no roundoff error:

$$x = a + i/2^M$$

as was claimed in the lemma.

We can now assume that $n \neq 0$ and $|x| \geq 2^{-M+1}$. We will show that the floating point number y computed in step $3d$ must satisfy:

$$y \in [2^{-B-2}, 2^{-M+1}]$$

Recall first from the definition of precomputed constants that

$$ln2a = \texttt{ln2\_a}[\texttt{hilo2}] = \ln(2) + s_x u 2^{-C-1}$$

for some $u \in [0, 1)$ and where s_x is, again, the sign of x (which comes in because of the index **hilo2**). Moreover, because of the fact that $ln2a$ is a $C + 1$ bit binary number, and n is a binary number of at most E bits, we see that the product $n * (-ln2a)$ can be computed exactly on the processor (using the fact that $E + C \leq F$). We can now begin our estimate of y:

$$\begin{aligned} y &= (x + n(-ln2a))(1 + su2^{-F}) \\ &= \left((n + r)(\ln(2) + s_x\epsilon_1) - su2^{-F}x - n(\ln(2) + s_x u2^{-C-1})\right)(1 + su2^{-F}) \\ &= rln2\_est + ns_x\left(\epsilon_1 + u2^{-C-1} - |x/n|su2^{-F}\right) \\ &\subset [2^{-B-1} - 2^{-C-1} - 2^{-F+1}, ln(2) + 2^{-B} + 2^{E-B} + 2^{E-C-1} + 2^{E-F+1}] \\ &\subset [2^{-B-2}, ln(2) + 1/8] \subset [2^{-B-2}, 7/8] \end{aligned}$$

where we have made use of the fact that $|x/n| \leq 2$, and the fact that $F \geq C \geq B + 3 > B \geq E + 4 > E \geq 11$, $-B \leq -15$, $E - B \leq -4$, $E - C - 1 \leq -5$, $E - F - 1 \leq -5$ and $\ln(2) < 3/4$.

In **step 3e** we find the integer i such that

$$y \in [\frac{i}{2^M}, \frac{i+1}{2^M}]$$

In **step 3f**, we adjust i so that if $y < 2^{-M}$ (or $n = 0$ which we considered earlier); otherwise, we decrement i so that

$$y \in [\frac{i}{2^M}, \frac{i+1}{2^M}) + 1/2^M$$

In **step 3e**, we subtract the (exactly representable) floating point number $i/2^M$ from y to get $a1$. In the case where $y < 2^{-M}$, then we must have

$$a1 = y - i/2^M \in [2^{-B-2}, 2^{-M})$$

If $y \geq 2^{-M}$, then we may have some error when making this subtraction, but since $a1 \geq 1/2^M$, this error is bounded by 2^{-F-M}, so we have:

$$a1 = (y - i/2^M)(1 - su2^{-F-M}) \in [2^{-M}, 2^{-M+1})$$

Finally, in **step 3f**, we add `n*(-ln2_b[hilo2])` to $a1$ to get a. In the case where $y \geq 1/2^M$, we know that $a1 > 2^{-M}$ and so

$$a \in [2^{-M} - 2^{E-C}, 2^{-M+1} + 2^{E-C}] \subset [2^{-M-1}, \frac{5}{4}2^{-M+1}]$$

where we have used the fact that $E + M + 1 < C$ to infer that $2^{E-C} < \frac{1}{2}2^{-M}$ In the case where $y < 2^{-M}$, we have

$$a \in [2^{-B-2}, 2^{-M}] + [-2^{E-C}, 2^{E-C}] \subset [0, 2^{-M+1}]$$

This proves the lemma.

Step 4 In this step we compute a Taylor series approximation z to $e^a - 1$. The analysis here is a little subtle because we must show that these few lines of code compute a true upper or lower bound (depending on `hilo`) for all $a \in [-2^{-M+1}, 2^{-M+1}]$. In step 4a, we set aa to be an upper or lower bound on $a * a$ (depending on `hilo`) We also set d to the smallest odd number with $d \geq expdeg$. Thus,

$$aa = a^2(1 + su2^{-F}) \;\; d \geq expdeg$$

for some $u \in [0, 1)$.

In **step 4b**, we compute the Horner rule term corresponding to the remainder term for the Taylor polynomial of degree d+1, this should be an estimate of $e^{\xi}_a a^{d+2}/(d+2)$ and since we assume $\xi \in [0, \ln(2)]$, we get:

$$e^{\xi}_a a^{d+2}/(d+2) \in 2^s a^{d+2}/(d+2)$$

where s is the rounding mode (1 for hi, -1 for lo). So, for some $u \in [0, 1)$ we have

$$e^{\xi}_a a^{d+2}/(d+2) = z(1 - s\epsilon) \;\; \epsilon = (2 - 1/2)a^{d+2}/d + 2$$

In **steps 4c,d** we compute the value of the Taylor polynomial with remainder using the Horner rule, by iterating the assignment:

$$z = ((z \oplus 1) \oslash (j \oplus 1) \otimes aa \oplus a) \oslash j;$$

Although it is not obvious, it is straightforward to check that as long as $z \in [0, 1)$ if this expression is evaluated with a rounding mode set to round down (RM), then the result will be a lower bound on the true value computed by the Horner rule, and if the rounding mode is set to round up (RP), then the result will be an upper bound. To see this, we consider the evaluation of this assignment using interval arithmetic given intervals for aa and z, and assuming that a and j are exact:

$$[z_-, z_+] \leftarrow (([z_-, z_+] \oplus 1) \oslash ([j,j] \oplus 1) \otimes [aa_-, aa_+] \oplus [a,a]) \oslash [j,j]$$

and using the fact that that aa_- and $z_- + 1$ will always be positive, we get

$$= [((z_- \underline{\oplus} 1) \underline{\oslash} (j \underline{\oplus} 1) \underline{\otimes} aa_- \underline{\oplus} a) \underline{\oslash} j, ((z_+ \overline{\oplus} 1) \bar{\oslash} (j \overline{\oplus} 1) \bar{\otimes} aa_- \overline{\oplus} a) \bar{\oslash} j]$$

which shows that we can compute lower bounds using RM and upper bounds using RP.

Next we estimate the error between the computed value of z and the actual value of $e^a - 1$.

Claim 3 *Let z_s be the floating point value computed in step 4, where as usual, s denotes the rounding mode, and s_a denotes the sign of a, then*

$$z_s = \phi^s((e^a - 1)(1 - s_a s 2^{2-M-F}))$$

and hence if $n = 0$, then $z_- < e^a < z_+$ and z_+ is at most 2 ULP larger than z_-.

Proof. [of claim] Let z_{d+2} denote the initial value assigned to z, and let z_j denote the value assigned to z inside the loop when the index variable has value j. Thus, for $j \in [1, d]$ we have

$$z_{2j-1} = ((z_{2j+1} + 1)/(2j) * aa + a)/(2j - 1);$$

Similarly, let $w_{d+2} = e^{\xi_a} a^{d+2}/(d+2)$ denote the actual remainder term in the (Horner form of the) Taylor polynomial for e^a, and define w_{2j-1} by

$$w_{2j} = (w_{2j+1} + 1)/(2j) * a^2 \quad w_{2j-1} = (w_{2j} + 1)/(2j - 1) * a$$

So that $e^a = 1 + w_1$, and we have

$$w_{2j-1} = ((w_{2j+1} + 1)/(2j) * a^2 + a)/(2j - 1)$$

We will compute an estimate for ϵ_j defined by $w_{2j+1} = z_{2j+1}(1 - s'\epsilon_{2j+1}$, where $s' = s s_z$ where s is the rounding mode and $s_z = s_a$ is the sign of z.

Let's estimate $z_{2j-1} - w_{2j-1}$.

$$\begin{aligned}
z_{2j-1} &= (((w_{2j+1}(1 - s'\epsilon)) \oplus 1) \oslash (2j) \otimes aa \oplus a) \oslash (2j - 1) \\
&= (((w_{2j+1}(1 - s'\epsilon_{2j+1})) + 1)(1 - su2^{-F}) \oslash (2j) \otimes aa \oplus a) \oslash (2j - 1) \\
&= (((w_{2j+1}(1 - s'\epsilon_{2j+1})) + 1)/(2j)(1 - su2^{-F})^2 \otimes aa \oplus a) \oslash (2j - 1) \\
&= (((w_{2j+1}(1 - s'\epsilon_{2j+1})) + 1)/(2j) * aa(1 - su2^{-F})^3 \oplus a) \oslash (2j - 1) \\
&= (((w_{2j+1}(1 - s'\epsilon_{2j+1})) + 1)/(2j) * a^2(1 - su2^{-F})^4 \oplus a) \oslash (2j - 1) \\
&= (((w_{2j+1}(1 - s'\epsilon_{2j+1})) + 1)/(2j) * a^2(1 - su2^{-F})^4 + a)/(2j - 1)(1 - s'u'2^{-F})^2 \\
&= (((w_{2j+1}) + 1)/(2j) * a^2(1 - s\epsilon'_{2j}) + a)/(2j - 1)(1 - s'u'2^{-F})^2
\end{aligned}$$

where

$$(1 - s\epsilon'_{2j}) = (1 - ss_z \epsilon_{2j+1} w_{2j+1}/(1 + w_{2j+1}))(1 - su2^{-F})^4$$

We can use the Taylor remainder to estimate $s_z w_{j+2}$ and we get

$$0 \leq s_z w_{2j+1} \leq 2s_z a^{2j+1}/(2j) \leq 22^{(2j+1)(1-M)}/(2j) \leq 2^{(2j+1)(1-M)}$$

Moreover we assume that $F \geq 52$ and $|a| < 2^{-M+1}$, so we get

$$\epsilon'_{2j} < \epsilon_{2j+1} 22^{(2j+1)(1-M)+1}$$

Continuing, we get

$$\begin{aligned} z_{2j-1} &= (w_{2j}(1 - s\epsilon'_{2j}) + a)/(2j-1)(1 - s'u'2^{-F})^2 \\ &= (w_{2j-1} - s\epsilon'_{2j} w_{2j}/(2j-1))(1 - su'2^{-F})^2 \\ &= w_{2j-1}(1 - s\epsilon'_{2j} w_{2j}/((2j-1)w_{2j-1})))(1 - su'2^{-F})^2 \end{aligned}$$

and we can estimate w_{2j}/w_{2j-1} by

$$w_{2j}/w_{2j-1} = e^{\xi_1} a^{2j}/(2j)/(e^{\xi_2} a^{2j-1}/(2j-1)) = e^{\xi_3} a(1 - 1/(2j))$$

where $|\xi_i| \leq |a| \leq 2^{1-M}$, so

$$s_z w_{2j}/w_{2j-1} \in [0, 2^{2-M}]$$

Thus, we get

$$\begin{aligned} z_{2j-1} &= w_{2j-1}(1 - ss_z \epsilon'_{2j} 2^{1-M})(1 - s'u'2^{-F})^2 \\ z_{2j-1} &= w_{2j-1}(1 - ss_z u\epsilon_{2j+1} 2^{(2j+1)(1-M)+2} 2^{1-M})(1 - s'u'2^{-F})^2 \\ z_{2j-1} &= w_{2j-1}(1 - ss_z u\epsilon_{2j+1} 2^{(2j+2)(1-M)+2})(1 - s'u'2^{-F})^2 \\ z_{2j-1} &= w_{2j-1}(1 - ss_z u\epsilon_{2j+1} 2^{(2j+2)(1-M)+2})(1 - s'u'2^{-F})^2 \\ &= w_{2j-1}(1 - s'\epsilon_{2j-1}) \end{aligned}$$

where

$$0 \leq \epsilon_{2j-1} = s'(1 - (1 - ss_z u\epsilon_{2j+1} 2^{(2j+2)(1-M)+2})(1 - s'u'2^{-F})^2)$$

Thus, we get a recurrence relation:

$$\epsilon_{2j-1} = s'(1 - (1 - s'u\epsilon_{2j+1} 2^{(2j+2)(1-M)+2})(1 - s'u'2^{-F})^2)$$

and if we let $d = 2j_d - 1$, with $j_d \geq 1$ then we have the initial condition that

$$\epsilon_{2j_d+1} \leq 3/22^{(1-M)(2j_d+1)}/(2j_d+1) \leq 2^{(1-M)(2j_d+1)-1}$$

Thus, as long as $(1 - M) \leq -2$ and $j \geq 1$ and $d \geq F/(M-1) + 2$ then

$$(1 - M) * (2j_d + 1) \leq (1 - M)(F/(M-1) + 2) \leq -F - 4$$

So

$$\epsilon_{2j_d+1} \leq 2^{(1-M)(2j_d+1)-1} \leq 2^{-F-5}$$

and

$$\epsilon_{2j-1} \leq s'(1 - (1 - s'u\epsilon_{2j+1}2^{-6})(1 - s'u'2^{-F})^2)$$

So by induction we get, for all j

$$\epsilon_{2j-1} \leq 32^{-F}$$

If we now analyze the last assignment separately, we have:

$$\begin{aligned} z_1 &= (((w_3(1 - s'\epsilon_3)) \oplus 1) \oslash (2) \otimes aa \oplus a) \oslash (1) \\ &= (((w_3(1 - s'\epsilon_3)) \oplus 1) \oslash (2) \otimes aa \oplus a \end{aligned}$$

So we can simplify the estimates and we get

$$z_2 = w_2(1 - s\epsilon_2) 0 \leq \epsilon_2 \leq \epsilon_3 2^{-6} \leq 2^{-F-4}$$

and

$$\begin{aligned} z_1 = \phi^s(a + z_2) &= \phi_s(a + w_2(1 - s2^{-F-4})) \\ &= \phi_s(w_1 - sw_2 2^{-F-4}) \\ &= \phi_s(w_1(1 - ss_z w_2/w_1 2^{-F-4}) \\ &= \phi_s(w_1(1 - s'2^{2-M-F})) \end{aligned}$$

Thus, z_1 is at most $1 + 2^{2-M}$ULP away from $e^a - 1$. In the case that $n = 0$ this proves the lemma. In the case where $n \neq 0$ we must still combine z with our estimates of $e^{i/2^m}$ which is done in step 5. $\square$ [of claim]

In **step 5**, we combine the computed value $1+z$ for e^a with the precomputed bound on $e^{i/2^M}$ which is represented as a sum $ea + eb$ of two floating point numbers. The fact that soundness is preserved by step 5 is easy to see. What is more subtle is that the mathematical error between v and e^x is at most 2 ULP(v). Let v' be the real number defined by

$$v' = (ea + zea + eb + zeb)2^n$$

Then,

$$
\begin{aligned}
e^x &= e^{a+i/2^M+n\ln(2)-su_1 2^{-F-M+1}} \\
&= e^a e^{i/2^M} 2^n e^{-su_1 2^{-F-M+1}} \\
&= (1+z)(1-su_2 2^{-F-M+1})(ea+eb-su_3 2^{-F-C})2^n e^{-su_1 2^{-F-M+1}} \\
&= (ea+zea+eb+zeb)2^n(1-su_4 2^{-F-M+3}) \\
&= v'(1-su_4 2^{-F-M+3})
\end{aligned}
$$

where the $u_i \in [0,1)$. Let v_s denote the value of the program variable v when the rounding mode is s. By the lemma on the evaluation of decreasing sums, we see that $v_- = \phi^-(v')$ and $v_+ = \phi_+(v')$. Thus, combining these observation, we see that if $M \geq 4$, then either $v_- = \phi^-(e^x)$ unless $v' < \sigma(v_-) < e^x$. Similarly, $v_+ = \phi^+(e^x)$ unless $e^x < \sigma^{-1}(v_+) < v'$ is less than Since the relative distance between them v' and e^x is less than $1/2$ ULP, we see that if they are on opposite sides of a floating point number v_0, then that number is the only number between them. In this case we must have $\sigma(v_-) = v_0 = \sigma^{-1}(v_+)$, so the upper and lower bounds differ by at most 2 ULP. From our argument we see that this happens only when e^x is within 2^{3-M} ULP of a floating point number. □ [of

Proposition]