Digit Serial Methods with Applications to Division and Square Root

(with mechanically checked correctness proofs)

Warren E. Ferguson Jr[1], Jesse Bingham[2], Levent Erkök[2], John R. Harrison[2] and Joe Leslie-Hurd[2]

[1]Intel Corporation, retired.
[2]Intel Corporation, Hillsboro, Oregon.

August 2, 2017

Abstract

We present a generic digit serial method (DSM) to compute the digits of a real number V. Bounds on these digits, and on the errors in the associated estimates of V formed from these digits, are derived. To illustrate our results, we derive such bounds for a parameterized family of high-radix algorithms for division and square root. These bounds enable a DSM designer to determine, for example, whether a given choice of parameters allows rapid formation and rounding of its approximation to V. All our claims are mechanically verified using the HOL-Light theorem prover, and are included in the appendix with commentary.

Keywords Digit serial method, digit recurrence method, on-the-fly technique, high-radix, division, square root, digit bounds, error bounds, formal verification, HOL Light.

1 Introduction

Let V be a real number. A digit serial method (DSM) is an algorithm that determines the digits of V serially, starting with the leading digit. A DSM begins by initializing an accumulator to zero and, as each digit is determined, that digit is aligned and added to the accumulator. Successive values of this accumulator form a sequence of estimates of V.

The primary contribution of this paper is a generic DSM analysis method for determining bounds on the magnitudes of the digits, as well as bounds on the error associated with the estimates of V. These bounds allow a designer to determine the required bit-width of signals representing these digits and errors, and to determine when the estimates of V can be efficiently formed and rounded by, say, on-the-fly techniques [7, 8].

The major results presented here are the Proxy Theorem 5.1 and its Corollary 5.3 with illustrations of their application to division and square root algorithms. These results have been checked/formalized using the HOL Light [12] theorem prover; a short extract from the formalization is presented in the appendix.

The analysis of low-radix DSM for division and square root is well-understood [9]. Analyses of specific high-radix DSM for these operations are described in [3, 11, 15]. An additional contribution of this paper is the application of our generic DSM analysis to a parameterized family of high-radix DSM algorithms for division and square root.

2 Scaling

The DSM considered here assume that $V \in (0, 1)$, so the leading digit of V is known to be the first fraction digit. For this assumption to be true, it may be necessary to scale the problem. Scaling is a three step process: (1) reduce the general problem to simpler problem by scaling, (2) determine the result of the simpler problem, and (3) reconstruct the general result from the result of the simpler problem.

For completeness, we briefly describe well-known scalings for division and square root of positive normalized finite precision binary floating-point numbers. Here, a positive normalized finite precision binary floating-point number is a real value of the form $s2^e$ composed of a normalized significand $s = 1 + f/2^k$, an integer exponent e, and a fraction $f/2^k$ where f is a non-negative integer less than 2^k for some positive integer k.

Scaling for division. Consider the computation of the quotient $Q \equiv (s_x 2^{e_x})/(s_y 2^{e_y})$ where s_x and s_y are normalized finite precision binary significands, and e_x and e_y are integers. Scaling reduces the computation of Q to the computation of a related quotient $V \in (0, 1)$, a DSM is used to compute V, and Q is reconstructed from the value of V. One possible scaling uses the reduction

$$V \equiv X/Y \quad \text{where} \quad (X, Y) \equiv (s_x/2, s_y),$$

so $X \in [1/2, 1)$, $Y \in [1, 2)$, and $V \in (1/4, 1)$. After the DSM determines V, the final result is reconstructed as follows:

$$Q = V 2^{e_x - e_y + 1}.$$

Scaling for square-root. Consider the computation of the square root $R \equiv \sqrt{s_x 2^{e_x}}$ where s_x is a normalized finite precision binary significand and e_x is an integer. Scaling reduces the computation of R to the computation of a related square root $V \in (0, 1)$, a DSM is used to compute V, and R is reconstructed from the value of V. One possible scaling uses the reduction

$$V \equiv \sqrt{X} \quad \text{where} \quad X \equiv \begin{cases} s_x/4 & \text{even } e_x \\ s_x/2 & \text{odd } e_x \end{cases},$$

so $X \in [1/4, 1)$ and $V \in [1/2, 1)$. After the DSM determines V, the final result is reconstructed as follows:

$$R = V \begin{cases} 2^{(e_x+2)/2} & \text{even } e_x \\ 2^{(e_x+1)/2} & \text{odd } e_x \end{cases} .$$

For both division and square root, scaling has reduced the original problem to the computation of a value $V \in (0, 1)$, combined with integer additions that determine the associated exponent.

3 Basic DSM

Consider the following mixed-radix representation of a real number V:

$$\begin{aligned} V &= \frac{1}{\beta_1}\Big(v_1 + \frac{1}{\beta_2}\Big(v_2 + \frac{1}{\beta_3}\Big(v_3 + \cdots\Big)\Big)\Big) \\ &= \frac{v_1}{B_1} + \frac{v_2}{B_2} + \frac{v_3}{B_3} + \cdots \end{aligned}$$

where[1] $\forall i \in \mathbb{N}^{>0} : B_i \equiv \beta_1\beta_2 \ldots \beta_i$. We always assume that $\{v_i\}_{i=1}^{\infty}$ is a sequence of integers (called *digits*), and that $\{\beta_i\}_{i=1}^{\infty}$ is a sequence of integers (called *radices* or *bases*), each 2 or greater. If $B_0 \equiv 1$, then $\forall i \in \mathbb{N} : B_{i+1} = \beta_{i+1}B_i$.

A DSM accumulates the terms of the series for V serially. Start with an accumulator initialized to 0. The terms involving the digits $v_1, v_2, v_3, \ldots$ are then consecutively added to the accumulator. The values of the accumulator after each digit is added defines the *head* sequence $\{H_i\}_{i=0}^{\infty}$ where:

$$H_0 \equiv 0,\ \forall i \in \mathbb{N}^{>0} : H_i \equiv \frac{v_1}{B_1} + \frac{v_2}{B_2} + \cdots + \frac{v_i}{B_i}.$$

Associated with each head H_i is the *tail* T_i defined as:

$$\forall i \in \mathbb{N} : T_i \equiv B_i\,(V - H_i) = B_i\left(\frac{v_{i+1}}{B_{i+1}} + \frac{v_{i+2}}{B_{i+2}} + \cdots\right).$$

Intuitively, H_i is the approximation to the target result V that has been computed after step i, while T_i is the error in this approximation normalized by B_i; here T_i/B_i is analogous to a floating-point value $s2^e$ with $T_i \sim s$ and $1/B_i \sim 2^e$. This definition of the tails provides the invariant $\forall i \in \mathbb{N} : V = H_i + T_i/B_i$.

We can summarize the above as follows:

$$\begin{aligned} &B_0 = 1,\ \forall i \in \mathbb{N} : B_{i+1} = \beta_{i+1}B_i, \\ &H_0 = 0,\ \forall i \in \mathbb{N} : H_{i+1} = H_i + v_{i+1}/B_{i+1},\ \text{and} \\ &T_0 = V,\ \forall i \in \mathbb{N} : T_{i+1} = \beta_{i+1}T_i - v_{i+1}. \end{aligned}$$

[1]Notation: Reals $\mathbb{R}$, non-negative reals $\mathbb{R}^{\geq 0}$, positive reals $\mathbb{R}^{>0}$, integers $\mathbb{Z}$, natural numbers $\mathbb{N} = \{0, 1, \ldots\}$, counting numbers $\mathbb{N}^{>0} = \{1, 2, \ldots\}$.

Digit selection. In the recurrence

$$\forall i \in \mathbb{N} : \beta_{i+1} T_i = v_{i+1} + T_{i+1}$$

note that

$$T_{i+1} = \frac{v_{i+2}}{\beta_{i+2}} + \frac{v_{i+3}}{\beta_{i+2}\beta_{i+3}} + \cdots$$

As we shall see in Sect. 4, if the digits satisfy $\forall k \geq 2 : |v_k| < \beta_k$, a simple algorithm can be used to accumulate the digits. If this condition holds then $|T_{i+1}|$, the distance between $\beta_{i+1} T_i$ and v_{i+1}, is at most 1. Consequently, a plausible choice for v_{i+1} is an integer near $\beta_{i+1} T_i$.

We therefore introduce *digit selection functions* $\forall i \in \mathbb{N}^{>0}$: $\mathrm{DSF}_i : \mathbb{R} \to \mathbb{Z}$ that "round" their real argument to a nearby integer, so $\forall i \in \mathbb{N} : v_{i+1} \equiv \mathrm{DSF}_{i+1}(\beta_{i+1} T_i)$. Paired with any digit selection function DSF is the *complementary digit selection function* coDSF : $\mathbb{R} \to \mathbb{R}$ defined as

$$\forall z \in \mathbb{R} : \mathrm{coDSF}(z) \equiv z - \mathrm{DSF}(z).$$

From the partition

$$\beta_{i+1} T_i = \mathrm{DSF}_{i+1}(\beta_{i+1} T_i) + \mathrm{coDSF}_{i+1}(\beta_{i+1} T_i)$$

of $\beta_{i+1} T_i$, we recognize that $T_{i+1} = \mathrm{coDSF}_{i+1}(\beta_{i+1} T_i)$.

Note that $|\mathrm{coDSF}(z)|$ is the distance between z and $\mathrm{DSF}(z)$, or equivalently the error in approximating z by $\mathrm{DSF}(z)$. It makes sense, then, to classify digit selection functions by the maximum value of $|\mathrm{coDSF}(z)|$ for all z.

Definition 3.1. (Round to Nearby Integer) For $\Omega \in \mathbb{R}$, $\mathrm{RNI}(\Omega)$ is the collection of all digit selection functions DSF : $\mathbb{R} \to \mathbb{Z}$ such that $\forall z \in \mathbb{R} : |\mathrm{coDSF}(z)| \leq \Omega$.

We argue that $\mathrm{RNI}(\Omega) = \emptyset$ when $\Omega < 1/2$. For suppose $\mathrm{RNI}(\Omega)$ is nonempty and choose $\mathrm{DSF} \in \mathrm{RNI}(\Omega)$. When $z = n + 1/2$ for some integer n, $\mathrm{DSF}(z)$ is an integer in the interval $[z - \Omega, z + \Omega]$. But that is impossible because there are no integers in this interval. Therefore, $\Omega < 1/2$ implies $\mathrm{RNI}(\Omega) = \emptyset$. For this reason we always assume that $\Omega \geq 1/2$.

When $\mathrm{DSF} \in \mathrm{RNI}(\Omega)$ with $1/2 \leq \Omega < 1$, $\mathrm{DSF}(z)$ belongs to the interval $[z - \Omega, z + \Omega]$, whose length 2Ω is in the interval $[1, 2)$. There is always one, and sometimes two, integers in this interval, and $\mathrm{DSF}(z)$ must be one of these integers.

Theorem 3.2. *If $v \equiv DSF(z)$ where $DSF \in RNI(\Omega)$, then $|v| \leq \lfloor |z| + \Omega \rfloor$.*

Proof. Since $\forall x : |\mathrm{coDSF}(x)| \leq \Omega$ and $v = \mathrm{DSF}(z)$, applying the triangle inequality yields

$$|v| = |\mathrm{DSF}(z)| = |z - \mathrm{coDSF}(z)| \leq |z| + \Omega.$$

The result follows by applying the floor function to the inequality and using the fact that v is an integer. □

Algorithm 1 Basic DSM that computes $\{(B_i, H_i, T_i)\}_{i=0}^{\infty}$ for $V \in \mathbb{R}$ where $\forall i \in \mathbb{N}^{>0} : (\text{DSF}_i \in \text{RNI}(\Omega_i)) \wedge (\beta_i \geq 2)$.

```
procedure DSM_BASIC(V)
  (B_0, H_0, T_0) := (1, 0, V)
  for i := 0, 1, 2, ... do
    {Invariant: V = H_i + T_i/B_i}
    v_{i+1} := DSF_{i+1}(β_{i+1} T_i)
    B_{i+1} := β_{i+1} B_i
    H_{i+1} := H_i + v_{i+1}/B_{i+1}
    T_{i+1} := β_{i+1} T_i − v_{i+1}
  end for
end procedure
```

Algorithm 1 is the result of combining the information presented above.[2] For this algorithm, bounds on the absolute error $|T_i|/B_i$ in the estimate H_i of V, and on the digit v_i, are easy to derive. We know that

$$|T_0| = V \quad \text{and} \quad \forall i \in \mathbb{N}^{>0} : |T_i| \leq \Omega_i$$

because $\forall i \in \mathbb{N}^{>0} : T_i = \text{coDSF}_i(\beta_i T_{i-1})$ where $\text{DSF}_i \in \text{RNI}(\Omega_i)$, and so applying Theorem 3.2 yields the digit bounds

$$\forall i \in \mathbb{N}^{>0} : |v_i| \leq \begin{cases} \lfloor \beta_1 V + \Omega_1 \rfloor & \text{if } i = 1 \\ \lfloor \beta_i \Omega_{i-1} + \Omega_i \rfloor & \text{if } i > 1 \end{cases}.$$

When the sequence $\{\Omega_i\}_{i=1}^{\infty}$ is bounded, so too is the tail sequence $\{T_i\}_{i=0}^{\infty}$. The following result proves that the head sequence converges to V if the tail sequence is bounded.

Theorem 3.3. *Let V and $\{\beta_i\}_{i=1}^{\infty}$ be given as described in Algorithm 1. If the sequence $\{T_i\}_{i=0}^{\infty}$ is bounded, then the sequence $\{H_i\}_{i=0}^{\infty}$ converges to V.*

Proof. Suppose the sequence $\{T_i\}_{i=0}^{\infty}$ is bounded, i.e., $\forall i \in \mathbb{N} : |T_i| \leq \Theta$ for some constant Θ. Because $\forall i \in \mathbb{N} : B_i \geq 2^i$, then $|T_i|/B_i \leq \Theta/2^i$ and therefore $\lim_{i\to\infty} T_i/B_i = 0$. Now $H_i = V - T_i/B_i$, and so

$$\begin{aligned} \lim_{i\to\infty} H_i &= \lim_{i\to\infty} (V - T_i/B_i) \\ &= \lim_{i\to\infty} V - \lim_{i\to\infty} T_i/B_i = V. \end{aligned}$$

□

4 On-the-fly Technique

When the on-the-fly technique applies, it offers an efficient way to accumulate the (integer) digits generated by a DSM. The binary on-the-fly technique can be described

[2] See the description of radix-conversion in [14].

$\beta_i A_{i-1}$	A_{i-1}				0	0	0	0
v_i	s	s	s	s	v	v	v	v
Sum when $s = 0$	A_{i-1}				v	v	v	v
Sum when $s = 1$	$A_{i-1} - 1$				v	v	v	v

Figure 1: 1-bit overlap; $\beta_i A_{i-1} + v_i$ when $\mu_i = 4$ and $s \equiv \text{signbit}(v_i)$.

as follows. We assume integers are represented using two's complement notation, and that $\forall i \in \mathbb{N}^{>0} : \beta_i \equiv 2^{\mu_i}$ where each $\mu_i \in \mathbb{N}^{>0}$.

First, no accumulation is needed to form $H_1 = v_1$, nor is there any restriction placed on the magnitude of v_1. Next, for $i \geq 2$, consider how the digit v_i is accumulated into H_{i-1} to form H_i:

$$H_i \equiv H_{i-1} + \frac{v_i}{B_i}.$$

Adding v_i/B_i to H_{i-1} creates a carry chain whose length can be nearly the bit-width of H_{i-1}. The goal of the on-the-fly technique is to eliminate this addition and its associated carry chain.

The simplest form of the on-the-fly technique assumes that $\forall i \geq 2 : |v_i| < \beta_i$, so both v_i and $v_i - 1$ have $(\mu_i + 1)$-bit two's complement representations. For each $i \geq 2$,

$$B_i H_i = B_i H_{i-1} + v_i = \beta_i B_{i-1} H_{i-1} + v_i$$

and so

$$A_i = \beta_i A_{i-1} + v_i$$

where $A_i \equiv B_i H_i$ is the accumulated value of all of the digits from v_1 through v_i, inclusively. Consider Figure 1 which illustrates the alignment of $\beta_i A_{i-1}$ and the sign-extended form of v_i when $\mu_i = 4$; note the 1 bit overlap between the leading (sign) bit of v_i and the trailing bit of A_{i-1}. When interpreted as a two's complement integer, the value of the bits of the sign-extended form of v_i that overlap A_{i-1} is either -1 or 0. From this observation we draw the following conclusions:

- when $v_i \in \mathbb{N}$: $s = 0$ and A_i is formed by concatenating the bits of A_{i-1} and the μ_i trailing bits of v_i, and
- when $v_i < 0$: $s = 1$ and A_i is formed by concatenating the bits of $A_{i-1} - 1$ with the μ_i trailing bits of v_i.

Consequently, if A_{i-1} and $A'_{i-1} \equiv A_{i-1} - 1$ are given, then A_i can be formed by appending the μ_i trailing bits of v_i to a selection of either A_{i-1} or A'_{i-1}. An analogous argument applies to the formation of $A'_i \equiv A_i - 1$ because

$$A'_i \equiv A_i - 1 = \beta_i A_{i-1} + v_i - 1 = \beta_i A_{i-1} + w_i$$

where we recall that $w_i \equiv v_i - 1$ also has a $(\mu_i + 1)$-bit two's complement representa-

$\beta_i A_{i-1}$	A_{i-1}				0	0	0	0
v_i	s	s	s	v	v	v	v	v

Figure 2: 2-bit overlap; $\beta_i A_{i-1} + v_i$ when $\mu_i = 4$ and $s \equiv \text{signbit}(v_i)$.

tion. In summary,

$$A_i = \begin{cases} \text{concatenate}(A_{i-1}, T_{\mu_i}(v_i)) & \text{if } v_i \in \mathbb{N} \\ \text{concatenate}(A'_{i-1}, T_{\mu_i}(v_i)) & \text{if } v_i < 0 \end{cases}, \text{ and}$$

$$A'_i = \begin{cases} \text{concatenate}(A_{i-1}, T_{\mu_i}(w_i)) & \text{if } w_i \in \mathbb{N} \\ \text{concatenate}(A'_{i-1}, T_{\mu_i}(w_i)) & \text{if } w_i < 0 \end{cases}.$$

where $T_\mu(z)$ consist of the trailing μ bits of the two's complement representation of the integer z.

This argument can be generalized in several ways. Consider, for example, the case where the digits cover the wider range $\forall i \geq 2 : |v_i| < 2\beta_i - 1$. In this case, because (μ_i+2)-bit two's complement integers range from $-2\beta_i$ to $2\beta_i - 1$ inclusively, each of the integers $\{v_i - 2, v_i - 1, v_i, v_i + 1\}$ has a $(\mu_i + 2)$-bit two's complement representation. Figure 2 illustrates the addition of one of these four integers to $\beta_i A_{i-1}$; note the 2-bit overlap between that integer and $\beta_i A_{i-1}$. The integer described by the bits in the overlap of the sign-extended form of the integer and $\beta_i A_{i-1}$ ranges from -2 to 1, inclusively. Therefore, because

$$A_i + k = \beta_i A_{i-1} + (v_i + k) \quad \text{for} \quad k \in \{-2, -1, 0, 1\}$$

we can form any one of the values $\{A_i - 2, A_i - 1, A_i, A_i + 1\}$ by adding the corresponding integer $\{v_i - 2, v_i - 1, v_i, v_i + 1\}$ to $\beta_i A_{i-1}$. For example, to form $A_i - 2$ add $z_i \equiv v_i - 2$ to $\beta_i A_{i-1}$. To perform this addition use the 2 leading bits of the $(\mu_i + 2)$-bit two's complement representation of z_i to select to which of $\{A_{i-1} - 2, A_{i-1} - 1, A_{i-1}, A_{i-1} + 1\}$ the trailing μ_i-bits of z_i are appended.

5 DSM Using a Proxy

Algorithm 1 is not effective for several reasons.

First, the value of V is used to initialize T_i. That's acceptable for recoding, where the algorithm converts the value of V in one form (say, binary) into another form (say, decimal). It's also acceptable in an analysis of the algorithm. It is not acceptable when actually performing a division or square root because it presupposes that the result of the computation is known before the algorithm starts.

Second, when the algorithm is applied to division or square root, the computation of the tails T_i involves a nontrivial division. For example, with the invariant written as $\forall i \in \mathbb{N} : T_i = B_i(V - H_i)$, it is simple to derive for the division problem $V \equiv X/Y$ that

$$\forall i \in \mathbb{N} : T_i Y = B_i(X - H_i Y),$$

Algorithm 2 DSM using a proxy that determines $\{(B_i, H_i, T_i)\}_{i=0}^{\infty}$ for $V \in \mathbb{R}^{\geq 0}$ where $\forall i \in \mathbb{N}^{>0} : (\text{DSF}_i \in \text{RNI}(\Omega_i)) \bigwedge (\beta_i \geq 2)$.

```
procedure DSM_PROXY(V, {ψ_i}_{i=0}^∞)
  (B_0, H_0, T_0) := (1, 0, V)
  for i := 0, 1, 2, ... do
    {Invariant: V = H_i + T_i/B_i}
    T_i^p := (1 + ψ_i)T_i
    v_{i+1} := DSF_{i+1}(β_{i+1} T_i^p)
    B_{i+1} := β_{i+1} B_i
    H_{i+1} := H_i + v_{i+1}/B_{i+1}
    T_{i+1} := β_{i+1} T_i − v_{i+1}
  end for
end procedure
```

and for the square root problem $V \equiv \sqrt{X}$ that

$$\forall i \in \mathbb{N} : T_i(V + H_i)/2 = B_i(X - H_i^2)/2.$$

In each of these equalities, the right-hand side can be computed via addition and multiplication of known finite precision values and the finite precision estimate H_i of V. However, given these right-hand sides, an unavoidable nontrivial division is required to determine the values of T_i.

Algorithm 1 determines the next digit v_{i+1} by approximately rounding $\beta_{i+1}T_i$ to an integer. It is plausible, then, that v_{i+1} can be determined using an accurate[3] proxy T_i^p for T_i. Algorithm 2 is a template for a DSM that uses a proxy T_i^p for T_i; it reduces to Algorithm 1 when $\forall i \in \mathbb{N} : \psi_i = 0$.

We make two assumptions about the proxies $\{T_i^p\}_{i=0}^{\infty}$.

- For analysis: The proxy T_i^p can be expressed as $T_i^p = (1 + \psi_i)T_i$; if $T_i \neq 0$ then $|\psi_i|$ is the relative error in the approximation of T_i by the proxy T_i^p.
- For implementation: The proxy T_i^p can be computed without knowledge of the exact values of V and T_i. When this assumption is satisfied, occurrences of V and T_i in Algorithm 2 can be eliminated. Examples of this elimination are presented in the following sections.

In Algorithm 2, the sequences $\{\text{DSF}_i\}_{i=1}^{\infty}$ and $\{\beta_i\}_{i=1}^{\infty}$ are considered to be fixed and to honor the restrictions stated in the header. We also suppose that ψ_i depends on V, T_i, and H_i; the dependence on H_i can be eliminated by applying the invariant $H_i = V - T_i/B_i$. In summary, T_{i+1} can be determined from just V and T_i.

To reduce the notational load, the dependence of T_i and T_i^p on V is represented implicitly.

Theorem 5.1. *(Proxy Theorem) In Algorithm 2 suppose that for some $V \in \mathbb{R}^{\geq 0}$ the sequence $\{\psi_i\}_{i=0}^{\infty}$ satisfies $\forall i \in \mathbb{N}, t \in \mathbb{R} : |\psi_i(V, t)| \leq \Psi_i(V, |t|)$ where Ψ_i is a*

[3] The accuracy of an approximation is measured by its relative error. The relative error of an approximation A' of $A \neq 0$ is $|\psi|$ where $A' = (1 + \psi)A$.

non-decreasing function of its second argument. Then for that V,

$$\forall i \in \mathbb{N} : (|T_i| \leq \tau_i(V)) \wedge (|T_i^p| \leq \tau_i^p(V))$$

where $\tau_i, \tau_i^p : \mathbb{R}^{\geq 0} \to \mathbb{R}$ are defined as

$$\begin{aligned}
&\tau_0(u) \equiv u,\\
&\forall i \in \mathbb{N} : \tau_{i+1}(u) \equiv \beta_{i+1}\Psi_i(u, \tau_i(u))\tau_i(u) + \Omega_{i+1}, \textit{ and}\\
&\forall i \in \mathbb{N} : \tau_i^p(u) \equiv (1 + \Psi_i(u, \tau_i(u)))\tau_i(u).
\end{aligned}$$

Proof. Suppose that $V \in \mathbb{R}^{\geq 0}$ and the sequence $\{\psi_i\}_{i=0}^{\infty}$ satisfies $\forall i \in \mathbb{N}, t \in \mathbb{R} : |\psi_i(V, t)| \leq \Psi_i(V, |t|)$ where Ψ_i is a non-decreasing function of its second argument.

We inductively prove that $\forall i \in \mathbb{N} : |T_i| \leq \tau_i(V)$ as follows. The base case is true $|T_0| = V = \tau_0(V)$. For the inductive step assume that $|T_i| \leq \tau_i(V)$ for some $i \in \mathbb{N}$. We know $T_i^p = (1 + \psi_i(V, T_i))T_i$, so application of the triangle inequality yields:

$$\begin{aligned}
|T_{i+1}| &= |\beta_{i+1}T_i - v_{i+1}|\\
&= |\beta_{i+1}T_i - \mathrm{DSF}_{i+1}(\beta_{i+1}T_i^p)|\\
&= |\beta_{i+1}T_i - (\beta_{i+1}T_i^p - \mathrm{coDSF}_{i+1}(\beta_{i+1}T_i^p))|\\
&= |\beta_{i+1}(T_i - T_i^p) + \mathrm{coDSF}_{i+1}(\beta_{i+1}T_i^p)|\\
&= |-\beta_{i+1}\psi_i(V, T_i)T_i + \mathrm{coDSF}_{i+1}(\beta_{i+1}T_i^p)|\\
&\leq \beta_{i+1}|\psi_i(V, T_i)||T_i| + \Omega_{i+1}.
\end{aligned}$$

Next, apply the assumption that $|\psi_i(V, t)| \leq \Psi_i(V, |t|)$, where Ψ_i is a non-decreasing function of its second argument, to continue this inequality as follows.

$$\begin{aligned}
|T_{i+1}| &\leq \beta_{i+1}|\psi_i(V, T_i)||T_i| + \Omega_{i+1}\\
&\leq \beta_{i+1}\Psi_i(V, |T_i|)|T_i| + \Omega_{i+1}\\
&\leq \beta_{i+1}\Psi_i(V, \tau_i(V))\tau_i(V) + \Omega_{i+1} \equiv \tau_{i+1}.
\end{aligned}$$

This completes the induction.

With the bounds on $\forall i \in \mathbb{N} : |T_i| \leq \tau_i(V)$ established, the bounds on $\forall i \in \mathbb{N} : |T_i^p|$ are obtained as follows. For each $i \in \mathbb{N}$:

$$\begin{aligned}
|T_i^p| &= |(1 + \psi_i(V, T_i))T_i|\\
&\leq (1 + |\psi_i(V, T_i)|)|T_i|\\
&\leq (1 + \Psi_i(V, |T_i|))|T_i|\\
&\leq (1 + \Psi_i(V, \tau_i(V)))\tau_i(V) \equiv \tau_i^p(V).
\end{aligned}$$

□

Definition 5.2. Let $\mathbb{P}$ be the subset of functions $\mathbb{R}^{>0} \to \mathbb{R}$ for which $p \in \mathbb{P}$ whenever $p(V)$ is a finite sum of terms of the form cV^n where $c \in \mathbb{R}^{\geq 0}$ and $n \in \mathbb{Z}$. ($\mathbb{P}$ is a subset of the posynomials in V [2, 6].)

Each $p \in \mathbb{P}$ is a convex function because on $\mathbb{R}^{>0}$ its second derivative is non-negative. Among the elements of $\mathbb{P}$ are each non-negative constant function as well as the identity function ν where $\forall u \in \mathbb{R}^{>0} : \nu(u) = u$. We also have these closure properties: for $p, q \in \mathbb{P}$ the functions p/ν, $p + q$, $pq \in \mathbb{P}$.

Corollary 5.3. *Let the assumptions of Theorem 5.1 hold for every $V \in \mathbb{R}^{>0}$. If*

$$\forall i \in \mathbb{N}, p \in \mathbb{P} : \Phi_i(p) \in \mathbb{P}.$$

where $\Phi_i : \mathbb{P} \to \mathbb{R}^{>0} \to \mathbb{R}$ is defined as

$$\forall p \in \mathbb{P}, u \in \mathbb{R}^{>0} : \Phi_i(p)(u) = \Psi_i(u, p(u)).$$

then for any closed subinterval $[a, b]$ of $\mathbb{R}^{>0}$,

$$\forall i \in \mathbb{N}, u \in [a, b]; \tau_i(u) \leq t_i \equiv \max(\tau_i(a), \tau_i(b)),$$
$$\forall i \in \mathbb{N}, u \in [a, b]; \tau_i^p(u) \leq t_i^p \equiv \max(\tau_i^p(a), \tau_i^p(b)).$$

Proof. Let the assumptions of this corollary hold. We first prove inductively that $\tau_i \in \mathbb{P}$ for each $i \in \mathbb{N}$. The base case is true because $\tau_0 = \nu \in \mathbb{P}$. For the inductive step let $\tau_i \in \mathbb{P}$ for some $i \in \mathbb{N}$. By assumption $\Phi_i(\tau_i) \in \mathbb{P}$, so by the closure properties $\tau_{i+1} = \beta_{i+1}\Phi_i(\tau_i)\tau_i + \Omega_{i+1} \in \mathbb{P}$, and this completes the inductive argument. Next, consider τ_i^p for any $i \in \mathbb{N}$. By assumption $\Phi_i(\tau_i) \in \mathbb{P}$ because $\tau_i \in \mathbb{P}$, so by the closure properties $\tau_i^p = (1 + \Phi_i(\tau_i))\tau_i \in \mathbb{P}$.

Let $[a, b]$ be a closed subinterval of $\mathbb{R}^{>0}$. Because functions in $\mathbb{P}$ are convex, we know that τ_i and τ_i^p attain their maximum on $[a, b]$ at either a or b. [16]. □

Combining the Theorem 3.2 with Corollary 5.3 yields for each $i \in \mathbb{N}$ and $V \in [a, b]$ that

$$|T_i| \leq t_i \quad \text{and} \quad |v_{i+1}| \leq \lfloor \beta_{i+1} t_i^p + \Omega_{i+1} \rfloor.$$

The formalization of the Proxy Theorem using the HOL Light theorem prover is presented in the appendix.

6 DSM for Division

As discussed in section 2, we consider the computation of $V \equiv X/Y$ where $X \in [1/2, 1)$ and $Y \in [1, 2)$. Algorithm 3 is an effective DSM that computes V; it uses an approximation $g(Y)$ of $1/Y$ obtained from, say, a lookup table. (Microprocessors often have an approximate reciprocal instruction.) The relative error in this approximation at Y is $|\sigma(Y)|$ where $\sigma : [1, 2) \to \mathbb{R}$ is defined so that

$$\forall Y \in [1, 2) : g(Y) \equiv (1 + \sigma(Y))/Y.$$

We assume $\forall Y \in [1, 2) : |\sigma(Y)| \leq \Sigma$ for some constant Σ.

Algorithm 3 DSM using a proxy for division that determines $\{(B_i, H_i, R_i)\}_{i=0}^{\infty}$ where $X \in [1/2, 1)$, $Y \in [1, 2)$, $V \equiv X/Y$, and $\forall i \in \mathbb{N}^{>0} : (\text{DSF}_i \in \text{RNI}(\Omega_i)) \bigwedge (\beta_i \geq 2)$.

```
procedure DSM_DIV(X, Y)
  (B_0, H_0, R_0) := (1, 0, X)
  for i := 0, 1, 2, ... do
    {Invariant: X = H_i Y + R_i / B_i}
    T_i^p := g(Y) R_i
    v_{i+1} := DSF_{i+1}(β_{i+1} T_i^p)
    B_{i+1} := β_{i+1} B_i
    H_{i+1} := H_i + v_{i+1} / B_{i+1}
    R_{i+1} := β_{i+1} R_i − v_{i+1} Y
  end for
end procedure
```

Reintroduce into Algorithm 3 the recursive computation of T_i as in Algorithm 2, and with it the invariant $\forall i \in \mathbb{N} : V = H_i + T_i/B_i$. As described in section 5, from this invariant we find that

$$\forall i \in \mathbb{N} : T_i Y = \underbrace{B_i(X - H_i Y)}_{\tilde{R}_i}.$$

The $\tilde{R}_i$ are called *partial remainders* for division and admit, for all $i \in \mathbb{N}$, the identity:

$$\begin{aligned}\tilde{R}_{i+1} - \beta_{i+1}\tilde{R}_i &= B_{i+1}(X - H_{i+1}Y) - \beta_{i+1}B_i(X - H_iY)\\ &= -B_{i+1}(H_{i+1} - H_i)Y\\ &= -v_{i+1}Y.\end{aligned}$$

We conclude that the partial remainders $\tilde{R}_i$ form one solution of the recurrence

$$\begin{aligned}&\tilde{R}_0 = X,\\ &\forall i \in \mathbb{N} : \tilde{R}_{i+1} = \beta_{i+1}\tilde{R}_i - v_{i+1}Y.\end{aligned}$$

The R_i computed by Algorithm 3 form another solution of this recurrence. Because this recurrence has a unique solution, we conclude that $\forall i \in \mathbb{N} : \tilde{R}_i = R_i$.

The approximate identity $g(Y)Y \approx 1$ allows division by Y to be replaced, approximately, by multiplication by $g(Y)$. Recall that $\forall i \in \mathbb{N} : T_iY = R_i$, so the proxy T_i^p for T_i is

$$\forall i \in \mathbb{N} : T_i^p \equiv g(Y)R_i.$$

A short computation shows that

$$\forall i \in \mathbb{N} : T_i^p = g(Y)R_i = g(Y)YT_i = (1 + \sigma(Y))T_i,$$

so the Proxy Theorem 5.1 can be applied with $\forall i \in \mathbb{N} : \psi_i(V, t) \equiv \sigma(Y)$ and $\forall i \in \mathbb{N} : \Psi_i(V, \tau) \equiv \Sigma$ because

$$\forall i \in \mathbb{N} : |\psi_i(V, t)| \equiv |\sigma(Y)| \leq \Sigma \equiv \Psi_i(V, |t|).$$

Algorithm 4 DSM using a proxy for square root that determines $\{(B_i, H_i, R_i)\}_{i=0}^{\infty}$ where $X \in [1/4, 1)$, $V \equiv \sqrt{X}$, and $\forall i \in \mathbb{N}^{>0} : (\text{DSF}_i \in \text{RNI}(\Omega_i)) \bigwedge (\beta_i \geq 2)$.

procedure DSM_SQRT(X)
  $(B_0, H_0, R_0) := (1, 0, X/2)$
  for $i := 0, 1, 2, \ldots$ **do**
    {**Invariant**: $X = H_i^2 + 2R_i/B_i$}
    $T_i^p := \mu_i g(X) R_i$
    $v_{i+1} := \text{DSF}_{i+1}(\beta_{i+1} T_i^p)$
    $B_{i+1} := \beta_{i+1} B_i$
    $H_{i+1} := H_i + v_{i+1}/B_{i+1}$
    $R_{i+1} := \beta_{i+1} R_i - v_{i+1}(H_{i+1} + H_i)/2$
  end for
end procedure

Clearly $\forall i \in \mathbb{N}, p \in \mathbb{P} : \Phi_i(p) = \Sigma \in \mathbb{P}$, so Corollary 5.3 applies. We conclude that $\forall i \in \mathbb{N} : |T_i| \leq t_i \equiv \tau_i(1)$ and $\forall i \in \mathbb{N} : |T_i^p| \leq t_i^p \equiv \tau_i^p(1)$ because each τ_i and τ_i^p is a non-negative increasing linear function on $[1/4, 1)$.

7 DSM for Square Root

As discussed in section 2, we consider the computation of $V \equiv \sqrt{X}$ for $X \in [1/4, 1)$. Algorithm 4 is an effective DSM that computes V; it uses an approximation $g(X)$ of $1/\sqrt{X}$. (Microprocessors often have an approximate reciprocal square root instruction.) The relative error in this approximation at X is $|\sigma(X)|$ where $\sigma : [1/4, 1) \to \mathbb{R}$ is defined so that

$$\forall X \in [1/4, 1) : g(X) \equiv (1 + \sigma(X))/\sqrt{X}.$$

We assume $\forall X \in [1/4, 1) : |\sigma(X)| \leq \Sigma$ for some constant Σ.

Reintroduce into Algorithm 4 the recursive computation of T_i as in Algorithm 2, and with it the invariant $\forall i \in \mathbb{N} : V = H_i + T_i/B_i$. As described in section 5, from this invariant we find that

$$\forall i \in \mathbb{N} : T_i(V + H_i)/2 = \underbrace{B_i(X - H_i^2)/2}_{\tilde{R}_i}.$$

The $\tilde{R}_i$ are called *partial remainders* for square root and admit, for all $i \in \mathbb{N}$, the identity:

$$\begin{aligned}\tilde{R}_{i+1} - \beta_{i+1}\tilde{R}_i &= B_{i+1}\frac{X - H_{i+1}^2}{2} - \beta_{i+1}B_i\frac{X - H_i^2}{2} \\ &= -B_{i+1}(H_{i+1} - H_i)\frac{H_{i+1} + H_i}{2} \\ &= -v_{i+1}\frac{H_{i+1} + H_i}{2}.\end{aligned}$$

We conclude that the partial remainders $\tilde{R}_i$ form one solution of the recurrence

$$\tilde{R}_0 = X/2,$$
$$\forall i \in \mathbb{N} : \tilde{R}_{i+1} = \beta_{i+1}\tilde{R}_i - v_{i+1}(H_{i+1} + H_i)/2.$$

The R_i computed by Algorithm 4 form another solution of this recurrence. Because this recurrence has a unique solution, we conclude that $\forall i \in \mathbb{N} : \tilde{R}_i = R_i$.

The proxy T_i^p for T_i is obtained by dividing R_i by an approximation of $(V+H_i)/2$. We argue that the approximate identity $\forall i \in \mathbb{N} : \mu_i g(X)(V + H_i)/2 \approx 1$ holds where

$$\mu_i \equiv (\text{if } i = 0 \text{ then } 2 \text{ else } 1)$$

because $g(X)V \approx 1$, $H_0 = 0$, and we expect $\forall i \in \mathbb{N}^{>0} : H_i \approx V$. This approximate identity allows division by $(V + H_i)/2$ to be replaced with multiplication by $\mu_i g(X)$, so the proxy T_i^p for T_i is

$$\forall i \in \mathbb{N} : T_i^p \equiv \mu_i g(X)R_i.$$

(The invariant tells us that $T_i = 2B_i V$ when $(V + H_i)/2 = 0$.)

Let $X \in [1/4, 1)$ be fixed, so $V \equiv \sqrt{X} \in [1/2, 1)$. For any $i \in \mathbb{N}$ we know $(V + H_i)/2 = V - T_i/(2B_i) = V(1 - T_i/(2VB_i))$ and $g(X)V = 1 + \sigma(X)$, so

$$\begin{aligned}
T_i^p &\equiv \mu_i g(X)R_i \\
&= \mu_i g(X)((V + H_i)/2)T_i \\
&= \mu_i g(X)V(1 - T_i/(2VB_i))T_i \\
&= \mu_i(1 + \sigma(X))(1 - T_i/(2VB_i))T_i.
\end{aligned}$$

Therefore, $T_i^p = (1 + \psi_i(V, T_i))T_i$ where

$$\psi_i(V, t) \equiv \sigma(X) - \begin{cases} 0 & \text{if } i = 0 \\ (1 + \sigma(X))(t/(2VB_i)) & \text{if } i > 0 \end{cases}$$

because $\mu_0(1 - T_0/(2VB_0)) = 1$, and so the Proxy Theorem 5.1 can be applied using

$$\Psi_i(V, |t|) \equiv \Sigma + \begin{cases} 0 & \text{if } i = 0 \\ (1 + \Sigma)(|t|/(2VB_i)) & \text{if } i > 0 \end{cases}.$$

Note that the first term Σ also occurs in Ψ_i for division. Clearly $\forall i \in \mathbb{N}, p \in \mathbb{P} : \Phi_i(p) \in \mathbb{P}$, so Corollary 5.3 applies and we conclude that $|T_i| \leq t_i \equiv \max(\tau_i(1/2), \tau_i(1))$ and $|T_i^p| \leq t_i^p \equiv \max(\tau_i^p(1/2), \tau_i^p(1))$.

8 Application

The results displayed in Table 1 describe the evolution of the bounds on the tails, tail proxies, and digits for the DSM algorithms for division and square root presented in the previous two sections. In this table the reciprocal and reciprocal root approximations are characterized by $\Sigma \equiv 2^{-9}$, and all digit selection functions belong to RNI(Ω) for

Table 1: DSM (using a proxy) for Division and Square Root with $\Sigma = 2^{-9}$ and $\Omega = 5/8$.

			Division			Digit		V = 1/4			V = 1	
i	$log_2(\beta_i)$	β_i	B_i	t_i	t_i^p	Bound	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$
0			1.00E+00	1.0000	1.0020		0.2500	0.0020	0.2505	1.0000	0.0020	1.0020
1	7	128	1.28E+02	0.8750	0.8767	128	0.6875	0.0020	0.6888	0.8750	0.0020	0.8767
2	7	128	1.64E+04	0.8438	0.8454	112	0.7969	0.0020	0.7984	0.8438	0.0020	0.8454
3	7	128	2.10E+06	0.8359	0.8376	108	0.8242	0.0020	0.8258	0.8359	0.0020	0.8376
4	7	128	2.68E+08	0.8340	0.8356	107	0.8311	0.0020	0.8327	0.8340	0.0020	0.8356

			Division			Digit		V = 1/4			V = 1	
i	$log_2(\beta_i)$	β_i	B_i	t_i	t_i^p	Bound	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$
0			1.00E+00	1.0000	1.0020		0.2500	0.0020	0.2505	1.0000	0.0020	1.0020
1	7	128	1.28E+02	0.8750	0.8767	128	0.6875	0.0020	0.6888	0.8750	0.0020	0.8767
2	5	32	4.10E+03	0.6797	0.6810	28	0.6680	0.0020	0.6693	0.6797	0.0020	0.6810
3	7	128	5.24E+05	0.7949	0.7965	87	0.7920	0.0020	0.7935	0.7949	0.0020	0.7965
4	7	128	6.71E+07	0.8237	0.8253	102	0.8230	0.0020	0.8246	0.8237	0.0020	0.8253

			Square Root			Digit		V = 1/2			V = 1	
i	$log_2(\beta_i)$	β_i	B_i	t_i	t_i^p	Bound	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$
0			1.00E+00	1.0000	1.0020		0.5000	0.0020	0.5010	1.0000	0.0020	1.0020
1	7	128	1.28E+02	0.8750	0.8797	128	0.7500	0.0078	0.7559	0.8750	0.0054	0.8797
2	7	128	1.64E+04	1.3761	1.3789	113	1.3761	0.0020	1.3789	1.2273	0.0020	1.2298
3	7	128	2.10E+06	0.9838	0.9858	177	0.9838	0.0020	0.9858	0.9377	0.0020	0.9396
4	7	128	2.68E+08	0.8710	0.8727	126	0.8710	0.0020	0.8727	0.8595	0.0020	0.8611

			Square Root			Digit		V = 1/2			V = 1	
i	$log_2(\beta_i)$	β_i	B_i	t_i	t_i^p	Bound	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$	$\tau_i(V)$	$\Phi_i(\tau_i)(V)$	$\tau_i^p(V)$
0			1.00E+00	1.0000	1.0020		0.5000	0.0020	0.5010	1.0000	0.0020	1.0020
1	7	128	1.28E+02	0.8750	0.8797	128	0.7500	0.0078	0.7559	0.8750	0.0054	0.8797
2	5	32	4.10E+03	0.8128	0.8145	28	0.8128	0.0022	0.8145	0.7756	0.0020	0.7772
3	7	128	5.24E+05	0.8489	0.8505	104	0.8489	0.0020	0.8505	0.8283	0.0020	0.8299
4	7	128	6.71E+07	0.8374	0.8390	109	0.8374	0.0020	0.8390	0.8322	0.0020	0.8338

$\Omega \equiv 5/8$. (The PN^2 or PNQ recoders discussed in [5] provide such digit selection functions.)

The table displays results for two choices of β-sequence:

- $\{\beta_1, \beta_2, \beta_3, \beta_4\} \equiv \{2^7, 2^7, 2^7, 2^7\}$, and
- $\{\beta_1, \beta_2, \beta_3, \beta_4\} \equiv \{2^7, 2^5, 2^7, 2^7\}$.

for each of division and square root. For each of these we obtain from Corollary 5.3, with ν the identity function, that

$$\begin{aligned}
&\tau_0 \equiv \nu, \\
&\forall i \in \mathbb{N} : \tau_{i+1} \equiv \beta_{i+1}\Phi_i(\tau_i)\tau_i + \Omega, \\
&\forall i \in \mathbb{N} : \tau_i^p \equiv (1 + \Phi_i(\tau_i))\tau_i
\end{aligned}$$

where for division

$$\forall \tau \in \mathbb{P} : \Phi_i(\tau) \equiv \Sigma$$

while for square root

$$\forall \tau \in \mathbb{P} : \Phi_i(\tau) \equiv \begin{cases} \Sigma & \text{if } \ i = 0 \\ \Sigma + (1+\Sigma)\tau/(2\nu B_i) & \text{if } \ i > 0 \end{cases} .$$

For any given value of V, we know the value of τ_0 and so we can compute $\Phi_0(\tau_0)(V)$ and then $\tau_0^p(V)$. This pattern is repeated for $i = 1, 2, 3, 4$ in succession; compute $\tau_i(V)$, then $\Phi_i(\tau_i)(V)$ and $\tau_i^p(V)$. From Corollary 5.3 we obtain

$$\begin{aligned}
&\forall i \in \mathbb{N}, V \in [a, b]; \tau_i(V) \le t_i \equiv \max(\tau_i(a), \tau_i(b)), \text{ and} \\
&\forall i \in \mathbb{N}, V \in [a, b]; \tau_i^p(V) \le t_i^p \equiv \max(\tau_i^p(a), \tau_i^p(b))
\end{aligned}$$

where $[a, b] \equiv [1/4, 1]$ for division and $[a, b] \equiv [1/2, 1]$ for square root. Finally, for $i = 1, 2, 3, 4$:

$$|T_i| \le t_i, \quad \text{and} \quad |v_i| \le \lfloor \beta_i t_{i-1}^p + \Omega \rfloor.$$

Observe that, for square root, the first β-sequence leads to an upper bound on $|T_2|$ that is larger than 1, and so the bound on $|v_3|$ is larger than $2^{\beta_3} = 2^7 = 128$. For the second β-sequence, obtained from the first β-sequence by decreasing β_2 from 2^7 to 2^5, we find that $|v_i| < 2^{\beta_i}$ for $2 \le i \le 4$ as well as $|T_4| < 1$; so the simplest form of on-the-fly accumulation of the digits can be applied. The reason why the reduction of β_2 from 2^7 to 2^5 is effective can be explained by the fact that

$$\Phi_1(\tau_1) = \Sigma + (1+\Sigma)\tau_1/(2\nu\beta_1)$$

and so

$$\tau_2 = \beta_2\Sigma + \Omega_2 + \beta_2(1+\Sigma)\tau_1/(2\nu\beta_1).$$

From the corresponding example for division we know $\beta_2\Sigma + \Omega_2 = 1/4 + 5/8 = 7/8$ when $\beta_2 = 2^7$. The third term contains the ratio β_2/β_1, so when β_2 is reduced from 2^7 to 2^5 the contribution of this third term is reduced by a factor of 4.

We performed additional experiments using a spreadsheet implementation of the DSM for division and square root [4] that expand on the results presented in Table 1. For specified values of the inputs (X and Y for division, X for square root), the spreadsheet computed the slack $s_i \equiv v_i^{max} - |v_i|$ where v_i^{max} is the upper bound on $|v_i|$ as discussed at the end of section 5. The spreadsheet's optimizer was used to determine inputs that made s_i small, i.e., made $|v_i|$ close to v_i^{max}. For both division and square root, and for each $i \in \{1, 2, 3, 4\}$, the optimizer was able to find inputs that made $|v_i|$ at least 96 percent of v_i^{max}.

9 Conclusion

The analysis presented in this paper is generic in the sense that no special properties of digit selection or reciprocal approximation are assumed. We have not considered how the digit selection function is implemented efficiently; we refer only to the references [4, 5, 10, 11, 15]. Nor have we discussed the effect of using one-sided approximations of the reciprocals, or biased digit selection functions.

The analysis presented here also extends to higher roots. For example, for the cube root $V = X^{1/3}$, from $T_i = B_i(V - H_i)$ it follows that

$$T_i(V^2 + VH_i + H_i^2)/3 = B_i(X - H_i^3)/3.$$

The partial remainders $R_i \equiv B_i(X - H_i^3)/3$ satisfy a two-term recurrence. Also, if $\nu_i = (\text{if i == 0 then 3 else 1})$ and $g(X) \approx X^{-2/3}$, then $T_i^p \equiv \nu_i g(X) R_i$ is a natural choice as the proxy for T_i because $\nu_i g(X)(V^2 + VH_i + H_i^2)/3 \approx 1$.

Prescaled division is also covered by the analysis presented here. Prescaled division computes $X' \equiv g(Y)X$ and $Y' \equiv g(Y)Y = 1 + \sigma(Y)$ before the for-loop; note that $X'/Y' = X/Y$. Inside the for-loop, the expressions

$$R_{i+1} = \beta_{i+1} R_i - v_{i+1} Y \quad \text{and} \quad X = H_i Y + R_i/B_i$$

for the partial remainder and the invariant become, after multiplication by $g(Y)$,

$$\begin{aligned} R'_{i+1} &= \beta_{i+1} R'_i - v_{i+1} Y' \\ &= (\beta_{i+1} R'_i - v_{i+1}) - v_{i+1}\sigma(Y), \text{ and} \\ X' &= H_i Y' + R'_i/B_i \end{aligned}$$

where $R'_i \equiv g(Y)R_i$. Note that $R'_0 \equiv X'$ and $T_i^p = R'_i$. The advantage of prescaled division is that, at a cost of two multiplications outside the for-loop, no multiplication inside the for-loop is needed to form the proxy T_i^p.

The proofs of the Proxy Theorem, its Corollary and the applications to division and square root, including verification of some concrete error bounds for particular instances, have been formally verified using the HOL Light theorem prover [12]; for the details see the appendix.

[4] For readers interested in replicting our results: These Excel 2016 spreadsheets are included as ancillary files `DSM_Division.xlsm` and `DSM_SquareRoot.xlsm`. The definition of the functions `dsf()`, `phidiv()`, and `phisqrt()` used in these spreadsheets are contained in a VBA Module. The optimization was performed by Excel's Solver Add-in using its Evolutionary mode of operation.

Acknowledgments

We thank Ping Tak Peter Tang, John O'Leary, Simon Rubanovich, and David Russinoff for many conversations related to the design and validation of floating-point arithmetic units.

A HOL Light proof of theorems

In this appendix, we discuss the full HOL Light [12] proof script for the claims made in the main body of the paper.

A.1 The main theorem 5.1

From this point on we present the actual ASCII proof script[5] required for HOL Light to prove the statements, interspersed with a few comments. Initially we load HOL Light's fairly extensive library of multivariate real and complex analysis. This is overkill for the relatively small amount of background material we need, but saves us from establishing from scratch various basic properties of convex functions. (In fact, a couple of additional properties of convex functions of general interest were added to the libraries as a direct result of supporting this proof.)

```
needs "Multivariate/realanalysis.ml";;
```

We now proceed to the main proof scripts. Note that HOL Light proof scripts are normally wrapped up in a `prove(assertion,tactics)` pair, but that the intermediate steps can be explored interactively via commands such as `g` (set goal) and `e` (expand current goal using tactics). For more information about the mechanics of HOL Light interaction see the tutorial [13]. Thus, the overall block for theorem 5.1 is an OCaml phrase binding to the desired name `THEOREM_V_1` the result of proving an assertion

```
let THEOREM_V_1 = prove
 (`!(V:real) (beta:num->real) (omega:num->real) (DSF:num->real->real)
    (B:num->real) (H:num->real) (v:num->real) (Tl:num->real) (Tp:num->real)
    (PSI:num->real#real->real) (psi:num->real#real->real)
    (tau:num->real->real) (taup:num->real->real).

        // Environmental assumptions including nondecreasing property
        &0 <= V /\
        (!i. i >= 0 ==> beta i > &0) /\
        (!i. i >= 1 ==> (!x. abs (x - DSF i x) <= omega i)) /\
        (!i. i >= 0 ==> abs (psi i (V,Tl i)) <= PSI i (V,abs(Tl i))) /\
        (!i x y. &0 <= x /\ x <= y ==> PSI i (V,x) <= PSI i (V,y)) /\

        (!u. tau 0 u = u) /\
        (!i u. tau (i + 1) u =
               beta (i + 1) * PSI i (u,tau i u) * tau i u + omega (i + 1)) /\
        (!i u. taup i u = (&1 + PSI i (u,tau i u)) * tau i u) /\

        // Computing recursively
        B 0 = &1 /\ H 0 = &0 /\ Tl 0 = V /\
```

[5] This HOL Light script is included as the ancillary file dsm.ml.

```
(!i. Tp i = (&1 + psi i (V,Tl i)) * Tl i) /\
(!i. v (i + 1) = DSF (i + 1) (beta (i + 1) * Tp i)) /\
(!i. B (i + 1) = beta (i + 1) * B i) /\
(!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
(!i. Tl (i + 1) = beta (i + 1) * Tl i - v (i + 1))

// Conclude loop invariant and bounds.
==> (!i. V = H i + Tl i / B i) /\
    (!i. i >= 0 ==> abs(Tl i) <= tau i V) /\
    (!i. i >= 0 ==> abs(Tp i) <= taup i V)`,
```

using the tactic script that follows, starting with some initial breakdown of the goal stripping off outer quantifiers and turning the antecedents of implications into assumptions of the goal state:

```
REPEAT GEN_TAC THEN REWRITE_TAC[GE; real_gt; real_gt; LE_0] THEN
STRIP_TAC THEN
```

We first establish by induction that all B_i are strictly positive:

```
SUBGOAL_THEN `!i:num. &0 < B i` ASSUME_TAC THENL
 [INDUCT_TAC THEN ASM_SIMP_TAC[REAL_LT_01; ADD1; REAL_LT_MUL];
  ALL_TAC] THEN
```

We then reshuffle the conjuncts to handle the τ^p clause first, assuming the other two clauses:

```
MATCH_MP_TAC(TAUT `(p /\ q ==> r) /\ p /\ q ==> p /\ q /\ r`) THEN
CONJ_TAC THENL
 [DISCH_THEN(STRIP_ASSUME_TAC o GSYM) THEN
  ASM_REWRITE_TAC[REAL_ABS_MUL] THEN GEN_TAC THEN
  MATCH_MP_TAC REAL_LE_MUL2 THEN ASM_REWRITE_TAC[REAL_ABS_POS] THEN
  MATCH_MP_TAC(REAL_ARITH `abs(x) <= a ==> abs(&1 + x) <= &1 + a`) THEN
  TRANS_TAC REAL_LE_TRANS `(PSI:num->real#real->real) i (V,abs(Tl i))` THEN
  ASM_SIMP_TAC[] THEN ASM_MESON_TAC[REAL_ABS_POS];
  ALL_TAC] THEN
```

Now we begin the main inductive proof and dispose of the base case by simple arithmetic:

```
REWRITE_TAC[AND_FORALL_THM] THEN
INDUCT_TAC THEN ASM_REWRITE_TAC[] THENL [ASM_REAL_ARITH_TAC; ALL_TAC] THEN
```

First we establish that the step case of the loop invariant holds

```
CONJ_TAC THENL
 [ASM_REWRITE_TAC[ADD1] THEN
  SUBGOAL_THEN `&0 < beta (i + 1) /\ &0 < B i` MP_TAC THENL
   [ASM_REWRITE_TAC[]; CONV_TAC REAL_FIELD];
  ALL_TAC] THEN
```

after which we massage the goal a little and chain through the inequalities, roughly following the paper proof:

```
FIRST_X_ASSUM(CONJUNCTS_THEN (ASSUME_TAC o GSYM)) THEN
REWRITE_TAC[ADD1] THEN

TRANS_TAC REAL_LE_TRANS
 `abs(-- beta (i + 1)  * psi i (V:real,Tl i) * Tl i +
      (beta (i + 1) * Tp i - DSF (i + 1) (beta (i + 1) * Tp i)))` THEN
CONJ_TAC THENL [ASM_REWRITE_TAC[] THEN REAL_ARITH_TAC; ALL_TAC] THEN
```

```
  TRANS_TAC REAL_LE_TRANS
   `beta (i + 1) * abs(psi i (V:real,Tl i)) * abs(Tl i) + omega(i + 1)` THEN
  CONJ_TAC THENL
   [MATCH_MP_TAC(REAL_ARITH
     `abs(x) <= a /\ abs(y) <= b ==> abs(x + y) <= a + b`) THEN
    ASM_SIMP_TAC[ARITH_RULE `1 <= i + 1`] THEN
    REWRITE_TAC[REAL_ABS_MUL; REAL_ABS_NEG] THEN
    ASM_SIMP_TAC[REAL_ARITH `&0 < x ==> abs x = x`; REAL_LE_REFL];
    ALL_TAC] THEN

  ASM_REWRITE_TAC[] THEN REWRITE_TAC[REAL_LE_RADD] THEN
  ASM_SIMP_TAC[REAL_LE_LMUL_EQ] THEN
  MATCH_MP_TAC REAL_LE_MUL2 THEN ASM_REWRITE_TAC[REAL_ABS_POS] THEN
  TRANS_TAC REAL_LE_TRANS `(PSI:num->real#real->real) i (V,abs(Tl i))` THEN
  ASM_SIMP_TAC[] THEN ASM_MESON_TAC[REAL_ABS_POS]);;
```

A.2 Properties of posynomials

The proof of corollary 5.3 requires a notion corresponding to a restricted subset of the posynomials, functions of V consisting of finite sums of positive multiples of integer powers of V, $\sum_1^k c_i V^{r_i}$. We render this in HOL Light as follows (using the simple word ‘posynomial’ is perhaps a little misleading since these are a restricted case, but this is only a name):

```
let posynomial = new_definition
 `posynomial p <=>
  ?c (n:num->real) k.
        (!i. 1 <= i /\ i <= k ==> c i > &0 /\ integer(n i)) /\
        (!v. &0 < v ==> sum (1..k) (\i. c i * v rpow (n i)) = p v)`;;
```

We now proceed to prove various basic ‘closure’ properties, roughly corresponding to those mentioned in the text. The identically zero function is a posynomial; even though the coefficients in the sum are assumed strictly positive, we can take $k = 0$ and get an empty sum:

```
let POSYNOMIAL_0 = prove
 (`posynomial (\v. &0)`,
  REWRITE_TAC[posynomial] THEN
  MAP_EVERY EXISTS_TAC [`(\i. &1):num->real`; `(\i. &0):num->real`; `0`] THEN
  REWRITE_TAC[SUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

Similarly straightforwardly, the identically 1 function is also a posynomial:

```
let POSYNOMIAL_1 = prove
 (`posynomial (\v. &1)`,
  REWRITE_TAC[posynomial] THEN
  MAP_EVERY EXISTS_TAC [`(\i. &1):num->real`; `(\i. &0):num->real`; `1`] THEN
  REWRITE_TAC[INTEGER_CLOSED; SUM_SING_NUMSEG; RPOW_POW] THEN REAL_ARITH_TAC);;
```

and indeed if p is a posynomial, so is any positive multiple of it

```
let POSYNOMIAL_CMUL = prove
 (`!p c. posynomial p /\ &0 < c ==> posynomial(\v. c * p(v))`,
  REPEAT GEN_TAC THEN
  DISCH_THEN(CONJUNCTS_THEN2 MP_TAC ASSUME_TAC) THEN
  REWRITE_TAC[posynomial] THEN DISCH_THEN(X_CHOOSE_THEN `d:num->real`
   (fun th -> EXISTS_TAC `(\i. c * d i):num->real` THEN MP_TAC th)) THEN
  REPEAT(MATCH_MP_TAC MONO_EXISTS THEN GEN_TAC) THEN
  SIMP_TAC[SUM_LMUL; GSYM REAL_MUL_ASSOC] THEN
  ASM_SIMP_TAC[real_gt; REAL_LT_MUL]);;
```

It is in fact convenient to record that any nonnegative constant function is a posynomial

```
let POSYNOMIAL_CONST = prove
 (`!c. &0 <= c ==> posynomial (\v. c)`,
  REWRITE_TAC[REAL_ARITH `&0 <= c <=> c = &0 \/ &0 < c`] THEN
  REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[POSYNOMIAL_0] THEN
  GEN_REWRITE_TAC (RAND_CONV o ABS_CONV) [GSYM REAL_MUL_RID] THEN
  MATCH_MP_TAC POSYNOMIAL_CMUL THEN
  ASM_REWRITE_TAC[POSYNOMIAL_1]);;
```

We next observe that multiplying a posynomial by an integer power of the variable again gives a posynomial:

```
let POSYNOMIAL_VPOWMUL = prove
 (`!p n. posynomial p /\ integer n ==> posynomial(\v. p(v) * v rpow n)`,
  REPEAT GEN_TAC THEN DISCH_THEN(CONJUNCTS_THEN2 MP_TAC ASSUME_TAC) THEN
  REWRITE_TAC[posynomial] THEN
  MATCH_MP_TAC MONO_EXISTS THEN X_GEN_TAC `c:num->real` THEN
  GEN_REWRITE_TAC BINOP_CONV [SWAP_EXISTS_THM] THEN
  MATCH_MP_TAC MONO_EXISTS THEN X_GEN_TAC `k:num` THEN
  DISCH_THEN(X_CHOOSE_THEN `nn:num->real` STRIP_ASSUME_TAC) THEN
  EXISTS_TAC `(\i. nn i + n):num->real` THEN
  ASM_SIMP_TAC[RPOW_ADD; REAL_MUL_ASSOC; SUM_RMUL; INTEGER_CLOSED]);;
```

This yields other basic closure properties as special cases: multiplying by V and dividing by V:

```
let POSYNOMIAL_VMUL = prove
 (`!p. posynomial p ==> posynomial(\v. p(v) * v)`,
  REPEAT STRIP_TAC THEN
  MP_TAC(ISPECL [`p:real->real`; `&1:real`] POSYNOMIAL_VPOWMUL) THEN
  ASM_REWRITE_TAC[RPOW_POW; REAL_POW_1; INTEGER_CLOSED]);;

let POSYNOMIAL_VDIV = prove
 (`!p. posynomial p ==> posynomial(\v. p(v) / v)`,
  REPEAT STRIP_TAC THEN
  MP_TAC(ISPECL [`p:real->real`; `-- &1:real`] POSYNOMIAL_VPOWMUL) THEN
  ASM_SIMP_TAC[RPOW_POW; real_div; RPOW_NEG; REAL_POW_1; INTEGER_CLOSED]);;
```

We can also trivially derive that the identity function is a posynomial:

```
let POSYNOMIAL_V = prove
 (`posynomial(\v. v)`,
  GEN_REWRITE_TAC (RAND_CONV o ABS_CONV) [GSYM REAL_MUL_LID] THEN
  MATCH_MP_TAC POSYNOMIAL_VMUL THEN REWRITE_TAC[POSYNOMIAL_1]);;
```

Slightly more involved is the fact that the sum of posynomials is a posynomial; note that following the strict form of the definition we need to plug two summations $1 \ldots n_1$ and $1 \ldots n_2$ into a single summation $1 \ldots n_1 + n_2$ with some straightforward but fiddly reindexing:

```
let POSYNOMIAL_ADD = prove
 (`!p q. posynomial p /\ posynomial q ==> posynomial(\v. p v + q v)`,
  REPEAT GEN_TAC THEN
  REWRITE_TAC[posynomial; IMP_CONJ; LEFT_IMP_EXISTS_THM] THEN
  MAP_EVERY X_GEN_TAC [`c1:num->real`; `n1:num->real`; `m:num`] THEN
  DISCH_TAC THEN DISCH_TAC THEN
  MAP_EVERY X_GEN_TAC [`c2:num->real`; `n2:num->real`; `n:num`] THEN
  DISCH_TAC THEN DISCH_TAC THEN
  EXISTS_TAC `\i. if i <= m then (c1:num->real) i else c2 (i - m)` THEN
  EXISTS_TAC `\i. if i <= m then (n1:num->real) i else n2 (i - m)` THEN
```

```
  EXISTS_TAC `m + n:num` THEN REWRITE_TAC[] THEN CONJ_TAC THENL
   [REPEAT STRIP_TAC THEN COND_CASES_TAC THEN ASM_SIMP_TAC[] THEN
    ASM_MESON_TAC[ARITH_RULE
     `~(i:num <= m) /\ i <= m + n ==> 1 <= i - m /\ i - m <= n`];
    REPEAT STRIP_TAC THEN ONCE_REWRITE_TAC[COND_RAND] THEN
    ONCE_REWRITE_TAC[MESON[] `(if p then f else g) (if p then x else y) =
        if p then f x else g y`] THEN
    SIMP_TAC[SUM_CASES; FINITE_NUMSEG; IN_NUMSEG;
      ARITH_RULE `(1 <= i /\ i <= m + n) /\ i <= m <=> 1 <= i /\ i <= m`;
      ARITH_RULE `(1 <= i /\ i <= m + n) /\ ~(i <= m) <=>
                  1 + m <= i /\ i <= n + m`] THEN
    REWRITE_TAC[GSYM numseg; SUM_OFFSET; ADD_SUB] THEN ASM_SIMP_TAC[]]);;
```

Now by induction we can establish that a finite sum of posynomials (based on some arbitrary indexing set k) is a posynomial:

```
let POSYNOMIAL_SUM = prove
 (`!k:A->bool p.
        FINITE k /\ (!i. i IN k ==> posynomial(\v. p v i))
        ==> posynomial (\v. sum k (p v))`,
  REWRITE_TAC[IMP_CONJ; RIGHT_FORALL_IMP_THM] THEN
  MATCH_MP_TAC FINITE_INDUCT_STRONG THEN
  SIMP_TAC[SUM_CLAUSES; POSYNOMIAL_0; POSYNOMIAL_ADD; FORALL_IN_INSERT;
           ETA_AX]);;
```

This yields without too much trouble the fact that the product of posynomials is a posynomial, simply by expanding the product of sums into a single sum over the Cartesian product of the indexing set (using HOL Light's standard theorem `SUM_SUM_PRODUCT`) and appealing to the just-proved `POSYNOMIAL_SUM`:

```
let POSYNOMIAL_MUL = prove
 (`!p q. posynomial p /\ posynomial q ==> posynomial(\v. p v * q v)`,
  REPEAT GEN_TAC THEN GEN_REWRITE_TAC (LAND_CONV o BINOP_CONV)
   [CONV_RULE (RAND_CONV(ONCE_DEPTH_CONV SYM_CONV)) (SPEC_ALL posynomial)] THEN
  STRIP_TAC THEN ASM_SIMP_TAC[posynomial] THEN
  REWRITE_TAC[GSYM posynomial] THEN
  SIMP_TAC[SUM_SUM_PRODUCT; FINITE_NUMSEG; REAL_MUL_SUM] THEN
  MATCH_MP_TAC POSYNOMIAL_SUM THEN
  SIMP_TAC[FINITE_PRODUCT_DEPENDENT; FINITE_NUMSEG; FORALL_IN_GSPEC] THEN
  REWRITE_TAC[IN_NUMSEG] THEN REPEAT STRIP_TAC THEN
  ONCE_REWRITE_TAC[REAL_ARITH
   `(c * x) * (d * y):real = (c * d) * (x * y)`] THEN
  SIMP_TAC[posynomial; GSYM RPOW_ADD] THEN REWRITE_TAC[GSYM posynomial] THEN
  MATCH_MP_TAC POSYNOMIAL_VPOWMUL THEN ASM_SIMP_TAC[INTEGER_CLOSED] THEN
  ONCE_REWRITE_TAC[GSYM REAL_MUL_RID] THEN
  RULE_ASSUM_TAC(REWRITE_RULE[real_gt]) THEN
  MATCH_MP_TAC POSYNOMIAL_CMUL THEN
  ASM_SIMP_TAC[REAL_LT_MUL; POSYNOMIAL_1]);;
```

Finally, we prove that each posynomial defines a convex function on the positive reals. (For more on convex functions see any standard book on convexity, e.g. [1] or [17].)

```
let REAL_CONVEX_ON_POSYNOMIAL = prove
 (`!p. posynomial p ==> p real_convex_on {x | x > &0}`,
  GEN_TAC THEN REWRITE_TAC[posynomial; LEFT_IMP_EXISTS_THM; real_gt] THEN
  MAP_EVERY X_GEN_TAC [`c:num->real`; `n:num->real`; `m:num`] THEN
  DISCH_THEN(CONJUNCTS_THEN2 ASSUME_TAC MP_TAC) THEN
  GEN_REWRITE_TAC (LAND_CONV o ONCE_DEPTH_CONV)
   [SET_RULE `&0 < v <=> v IN {x | &0 < x}`] THEN
  MATCH_MP_TAC(MESON[REAL_CONVEX_ON_EQ]
    `is_realinterval s /\ f real_convex_on s
     ==> (!x. x IN s ==> f x = g x) ==> g real_convex_on s`) THEN
```

```
  REWRITE_TAC[IS_REALINTERVAL_CLAUSES] THEN
  MATCH_MP_TAC REAL_CONVEX_ON_SUM THEN
  REWRITE_TAC[FINITE_NUMSEG; IN_NUMSEG] THEN
  X_GEN_TAC `i:num` THEN STRIP_TAC THEN MATCH_MP_TAC REAL_CONVEX_LMUL THEN
  ASM_SIMP_TAC[REAL_LT_IMP_LE] THEN
  MATCH_MP_TAC REAL_CONVEX_ON_RPOW_INTEGER THEN
  ASM SET_TAC[]);;
```

A.3 Corollary 5.3

We can now establish the corollary:

```
let COROLLARY_V_3 = prove
 (`!(V:real) (beta:num->real) (omega:num->real) (DSF:num->real->real)
    (B:num->real) (H:num->real) (v:num->real) (Tl:num->real) (Tp:num->real)
    (PSI:num->real#real->real) (psi:num->real#real->real)
    (tau:num->real->real) (taup:num->real->real).

        // Environmental assumptions including nondecreasing property
        &0 < V /\
        (!i. i >= 0 ==> beta i > &0) /\
        (!i. i >= 1 ==> (!x. abs (x - DSF i x) <= omega i)) /\
        (!i. i >= 0 ==> abs (psi i (V,Tl i)) <= PSI i (V,abs(Tl i))) /\
        (!i x y. &0 <= x /\ x <= y ==> PSI i (V,x) <= PSI i (V,y)) /\

        (!u. tau 0 u = u) /\
        (!i u. tau (i + 1) u =
               beta (i + 1) * PSI i (u,tau i u) * tau i u + omega (i + 1)) /\
        (!i u. taup i u = (&1 + PSI i (u,tau i u)) * tau i u) /\

        // Computing recursively
        B 0 = &1 /\ H 0 = &0 /\ Tl 0 = V /\

        (!i. Tp i = (&1 + psi i (V,Tl i)) * Tl i) /\
        (!i. v (i + 1) = DSF (i + 1) (beta (i + 1) * Tp i)) /\
        (!i. B (i + 1) = beta (i + 1) * B i) /\
        (!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
        (!i. Tl (i + 1) = beta (i + 1) * Tl i - v (i + 1)) /\

        // The extra posynomial-related assumption
        (!i p. i >= 0 /\ posynomial p
               ==> posynomial (\v. PSI i (v,p v)))

        // Hence conclude our bounds
        ==> !a b. real_interval[a,b] SUBSET {x | x > &0}
                  ==> !i u. u IN real_interval[a,b]
                            ==> tau i u <= max (tau i a) (tau i b) /\
                                taup i u <= max (taup i a) (taup i b)`,
```

by combining the original proxy theorem with some basic properties of posynomials. After some initial breakdown of the goal, also standardizing inequalities by writing $s > t$ as $t < s$ and so on, we make the trivial deduction $0 \leq V$ from the assumption $0 < V$ (to settle this in the hypotheses once and for all for convenient use without explicit mention):

```
  REWRITE_TAC[real_gt; real_ge; GT; GE; LE_0] THEN
  REPEAT GEN_TAC THEN STRIP_TAC THEN
  FIRST_ASSUM(ASSUME_TAC o MATCH_MP REAL_LT_IMP_LE) THEN
  REPEAT GEN_TAC THEN DISCH_TAC THEN
```

we first prove that each τ_i defines a posynomial, by induction:

```
  SUBGOAL_THEN `!i:num. posynomial (tau i)` ASSUME_TAC THENL
   [INDUCT_TAC THEN GEN_REWRITE_TAC RAND_CONV [GSYM ETA_AX] THEN
    ASM_REWRITE_TAC[ADD1; POSYNOMIAL_V] THEN
    MATCH_MP_TAC POSYNOMIAL_ADD THEN CONJ_TAC THENL
     [MATCH_MP_TAC POSYNOMIAL_CMUL THEN ASM_REWRITE_TAC[] THEN
      MATCH_MP_TAC POSYNOMIAL_MUL THEN ASM_SIMP_TAC[ETA_AX];
      MATCH_MP_TAC POSYNOMIAL_CONST THEN
      ASM_MESON_TAC[REAL_LE_TRANS; REAL_ABS_POS; ARITH_RULE `1 <= i + 1`]];
    ALL_TAC] THEN
```

and then, using that as a lemma, that the same is true of τ_i^p:

```
  SUBGOAL_THEN `!i:num. posynomial (taup i)` ASSUME_TAC THENL
   [INDUCT_TAC THEN GEN_REWRITE_TAC RAND_CONV [GSYM ETA_AX] THEN
    REWRITE_TAC[ADD1] THEN ONCE_ASM_REWRITE_TAC[] THEN
    MATCH_MP_TAC POSYNOMIAL_MUL THEN REWRITE_TAC[ETA_AX] THEN
    (CONJ_TAC THENL [ALL_TAC; FIRST_X_ASSUM MATCH_ACCEPT_TAC]) THEN
    MATCH_MP_TAC POSYNOMIAL_ADD THEN REWRITE_TAC[POSYNOMIAL_1] THEN
    FIRST_X_ASSUM MATCH_MP_TAC THEN REWRITE_TAC[ETA_AX] THEN
    FIRST_X_ASSUM MATCH_ACCEPT_TAC;
    ALL_TAC] THEN
```

The result then follows by appealing to a general bound property that the upper bound of a convex function on a real interval is attained at one of the endpoints (`REAL_CONVEX_LOWER_REAL_INTERVAL`) and the fact that posynomials are convex functions `REAL_CONVEX_ON_POSYNOMIAL` proved at the end of the previous section:

```
  REPEAT STRIP_TAC THEN
  MATCH_MP_TAC REAL_CONVEX_LOWER_REAL_INTERVAL THEN
  ASM_REWRITE_TAC[] THEN
  FIRST_X_ASSUM(MATCH_MP_TAC o MATCH_MP (REWRITE_RULE[IMP_CONJ_ALT]
        REAL_CONVEX_ON_SUBSET)) THEN
  REWRITE_TAC[GSYM real_gt] THEN MATCH_MP_TAC REAL_CONVEX_ON_POSYNOMIAL THEN
  FIRST_X_ASSUM MATCH_ACCEPT_TAC);;
```

Before proceeding, for convenience, we collect together a 'kitchen sink' version of the main proxy theorem and corollary together:

```
let FULL_COROLLARY = prove
 (`!(V:real) (beta:num->real) (omega:num->real) (DSF:num->real->real)
    (B:num->real) (H:num->real) (v:num->real) (Tl:num->real) (Tp:num->real)
    (PSI:num->real#real->real) (psi:num->real#real->real)
    (tau:num->real->real) (taup:num->real->real).

        // Environmental assumptions including nondecreasing property
        &0 < V /\
        (!i. i >= 0 ==> beta i > &0) /\
        (!i. i >= 1 ==> (!x. abs (x - DSF i x) <= omega i)) /\
        (!i. i >= 0 ==> abs (psi i (V,Tl i)) <= PSI i (V,abs(Tl i))) /\
        (!i x y. &0 <= x /\ x <= y ==> PSI i (V,x) <= PSI i (V,y)) /\

        (!u. tau 0 u = u) /\
        (!i u. tau (i + 1) u =
               beta (i + 1) * PSI i (u,tau i u) * tau i u + omega (i + 1)) /\
        (!i u. taup i u = (&1 + PSI i (u,tau i u)) * tau i u) /\

        // Computing recursively
        B 0 = &1 /\ H 0 = &0 /\ Tl 0 = V /\

        (!i. Tp i = (&1 + psi i (V,Tl i)) * Tl i) /\
        (!i. v (i + 1) = DSF (i + 1) (beta (i + 1) * Tp i)) /\
        (!i. B (i + 1) = beta (i + 1) * B i) /\
```

```
        (!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
        (!i. Tl (i + 1) = beta (i + 1) * Tl i - v (i + 1)) /\

        // The extra posynomial-related assumption
        (!i p. i >= 0 /\ posynomial p
               ==> posynomial (\v. PSI i (v,p v)))

        // Hence conclude invariant and all bounds.
        ==> (!i. V = H i + Tl i / B i) /\
            (!i. abs(Tl i) <= tau i V) /\
            (!i. abs(Tp i) <= taup i V) /\
            (!a b. real_interval[a,b] SUBSET {x | x > &0}
                   ==> !i u. u IN real_interval[a,b]
                             ==> tau i u <= max (tau i a) (tau i b) /\
                                 taup i u <= max (taup i a) (taup i b))`,
```

The proof is just a trivial if mildly tedious instantiation of earlier results; this could have been done in one piece at the outset, but we preserved the separate results from the earlier development:

```
  REWRITE_TAC[real_gt; real_ge; GT; GE; LE_0] THEN
  REPEAT GEN_TAC THEN STRIP_TAC THEN
  FIRST_ASSUM(ASSUME_TAC o MATCH_MP REAL_LT_IMP_LE) THEN
  ONCE_REWRITE_TAC[TAUT `p /\ q /\ r /\ s <=> (p /\ q /\ r) /\ s`] THEN
  CONJ_TAC THENL
   [MATCH_MP_TAC(REWRITE_RULE[GE; LE_0] THEOREM_V_1) THEN
    MAP_EVERY EXISTS_TAC
     [`beta:num->real`; `omega:num->real`; `DSF:num->real->real`;
      `v:num->real`; `PSI:num->real#real->real`;
      `psi:num->real#real->real`] THEN
    ASM_REWRITE_TAC[real_gt];

    MATCH_MP_TAC(REWRITE_RULE[real_gt] COROLLARY_V_3) THEN
    MAP_EVERY EXISTS_TAC
     [`V:real`; `beta:num->real`; `omega:num->real`; `DSF:num->real->real`;
      `B:num->real`; `H:num->real`; `v:num->real`; `Tl:num->real`;
      `Tp:num->real`;
      `PSI:num->real#real->real`; `psi:num->real#real->real`] THEN
    ASM_REWRITE_TAC[GE; LE_0]]);;
```

A.4 Instantiation to division (Section 6)

We next proceed with the instantiation to the special cases of division:

```
let BOUND_THEOREM_DIV = prove
 (`!beta Sigma omega B DSF H R Tp X Y g sigma v.
        (!i. i >= 0 ==> beta i > &0) /\
        &1 / &2 <= X /\ X < &1 /\
        &1 <= Y /\ Y < &2 /\
        (!y. &1 <= y /\ y < &2
             ==> g y = (&1 + sigma y) / y /\ abs(sigma y) <= Sigma) /\
        (!i. i >= 1 ==> (!x. abs (x - DSF i x) <= omega i)) /\
        B 0 = &1 /\ H 0 = &0 /\ R 0 = X /\
        (!i. Tp i = g(Y) * R i) /\
        (!i. v (i + 1) = DSF (i + 1) (beta (i + 1) * Tp i)) /\
        (!i. B (i + 1) = beta (i + 1) * B i) /\
        (!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
        (!i. R (i + 1) = beta (i + 1) * R i -  v(i + 1) * Y)
        ==> ?tau. (!u. tau 0 u = u) /\
                  (!i u. tau (i + 1) u =
                         beta (i + 1) * Sigma * tau i u + omega (i + 1)) /\
                  (!i. abs(X / Y - H i)
                       <= max (tau i (&1 / &4)) (tau i (&1)) / B i)`,
```

We begin by establishing a few obvious facts that we want to avoid re-proving later such as $0 < B_i$, and deducing that there are indeed functions τ and T satisfying the recursion equations in the proxy theorem:

```
REPEAT GEN_TAC THEN REWRITE_TAC[GE; LE_0; real_gt] THEN STRIP_TAC THEN
SUBGOAL_THEN `&0 <= Sigma` ASSUME_TAC THENL
 [FIRST_X_ASSUM(MP_TAC o SPEC `&1:real`) THEN REAL_ARITH_TAC;
  ALL_TAC] THEN
SUBGOAL_THEN `!i. &0 < (B:num->real) i` ASSUME_TAC THENL
 [INDUCT_TAC THEN ASM_SIMP_TAC[REAL_LT_MUL; ADD1; REAL_LT_01]; ALL_TAC] THEN
SUBGOAL_THEN `&0 < X /\ &0 < Y` STRIP_ASSUME_TAC THENL
 [ASM_REAL_ARITH_TAC; ALL_TAC] THEN
SUBGOAL_THEN `&0 < X / Y` ASSUME_TAC THENL
 [ASM_MESON_TAC[REAL_LT_DIV]; ALL_TAC] THEN
MAP_EVERY ABBREV_TAC
 [`PSI:num->real#real->real = \i (u,t). Sigma`;
  `psi:num->real#real->real = \i (u,t). sigma(Y:real)`] THEN
(X_CHOOSE_THEN `tau:num->real->real`
  (STRIP_ASSUME_TAC o REWRITE_RULE[ADD1]) o
 prove_recursive_functions_exist num_RECURSION)
  `(!u:real. tau 0 u = u) /\
   (!i u. tau (SUC i) u =
          beta (i + 1) * PSI i (u,tau i u) * tau i u + omega (i + 1))` THEN
(X_CHOOSE_THEN `Tl:num->real`
  (STRIP_ASSUME_TAC o REWRITE_RULE[ADD1]) o
 prove_recursive_functions_exist num_RECURSION)
  `Tl 0 :real = X / Y /\
  !i. Tl (SUC i) = beta (i + 1) * Tl i - v (i + 1)` THEN
ABBREV_TAC
 `taup:num->real->real = \i u. (&1 + PSI i (u,tau i u)) * tau i u` THEN
```

We then simply instantiate the proxy theorem/corollary appropriately:

```
MP_TAC(ISPECL
 [`X / Y:real`;
  `beta:num->real`;
  `omega:num->real`;
  `DSF:num->real->real`;
  `B:num->real`;
  `H:num->real`;
  `v:num->real`;
  `Tl:num->real`;
  `Tp:num->real`;
  `PSI:num->real#real->real`;
  `psi:num->real#real->real`;
  `tau:num->real->real`;
  `taup:num->real->real`]
  FULL_COROLLARY) THEN
```

Now after some trivial cleanup and splitting

```
REWRITE_TAC[GE; LE_0; real_gt] THEN ANTS_TAC THENL
```

we first need to verify the various hypotheses of the proxy theorem and corollary. In all we get 17(!) of them. However, it turns out that most have trivial one-line proofs like `FIRST_X_ASSUM MATCH_ACCEPT_TAC`. The only one with a little content is proving that `!i. Tp i = (&1 + psi i (X / Y,Tl i)) * Tl i`. After a little initial rearrangement this devolves to proving `!j. R j / Y = Tl j`, which is done by an easy induction (this corresponds to verifying the equivalence of R and $\tilde{R}$ in the text). Now we have the conclusions from the main theorem/corollary and we do some instantiation, in particular setting the endpoints of the interval for which the bound is derived, and hence derive our result:

```
    STRIP_TAC THEN EXISTS_TAC `tau:num->real->real` THEN
    ASM_REWRITE_TAC[REAL_ADD_SUB] THEN CONJ_TAC THENL
     [EXPAND_TAC "PSI" THEN REWRITE_TAC[]; ALL_TAC] THEN
    ASM_SIMP_TAC[REAL_ABS_DIV; REAL_LE_DIV2_EQ;
                 REAL_ARITH `&0 < b ==> abs b = b`] THEN
    X_GEN_TAC `i:num` THEN
    FIRST_X_ASSUM(MP_TAC o SPECL [`&1 / &4`; `&1`]) THEN
    REWRITE_TAC[SUBSET; IN_REAL_INTERVAL; IN_ELIM_THM] THEN
    ANTS_TAC THENL [REAL_ARITH_TAC; ALL_TAC] THEN
    DISCH_THEN(MP_TAC o SPECL [`i:num`; `X / Y:real`]) THEN
    ANTS_TAC THENL [ALL_TAC; ASM_MESON_TAC[REAL_LE_TRANS]] THEN
    REWRITE_TAC[REAL_ARITH
     `&1 / &4 <= X / Y /\ X / Y <= &1 <=>
      &1 / &2 * inv(&2) <= X * inv Y /\ X * inv Y <= &1 * inv(&1)`] THEN
    CONJ_TAC THEN MATCH_MP_TAC REAL_LE_MUL2 THEN REPEAT CONJ_TAC THEN
    TRY(MATCH_MP_TAC REAL_LE_INV2) THEN
    REWRITE_TAC[REAL_LE_INV_EQ] THEN ASM_REAL_ARITH_TAC]);;
```

A.5 Instantiation to square root (Section 7)

This is conceptually the same as the instantiation to division, but various terms become more involved and as a result the proof becomes a bit more complicated too.

```
let BOUND_THEOREM_SQRT = prove
 (`!beta Sigma omega B DSF H R Tp X g sigma v.
        (!i. i >= 0 ==> beta i > &0) /\
        &1 / &4 <= X /\ X < &1 /\
        (!x. &1 / &4 <= x /\ x < &1
             ==> g x = (&1 + sigma x) / sqrt x /\
                  abs(sigma x) <= Sigma) /\
        (!i. i >= 1 ==> (!x. abs (x - DSF i x) <= omega i)) /\
        B 0 = &1 /\ H 0 = &0 /\ R 0 = X / &2 /\
        (!i. Tp i = (if i = 0 then &2 else &1) * g(X) * R i) /\
        (!i. v (i + 1) = DSF (i + 1) (beta (i + 1) * Tp i)) /\
        (!i. B (i + 1) = beta (i + 1) * B i) /\
        (!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
        (!i. R (i + 1) =
             beta (i + 1) * R i -  v(i + 1) * (H(i + 1) + H i) / &2)
        ==> ?tau.
                  (!u. tau 0 u = u) /\
                  (!i u. tau (i + 1) u =
                         beta (i + 1) *
                         (if i = 0 then Sigma
                          else Sigma + (&1 + Sigma) * tau i u / (&2 * u * B i))
                          * tau i u +
                         omega (i + 1)) /\
                  (!i. abs(sqrt X - H i)
                       <= max (tau i (&1 / &2)) (tau i (&1)) / B i)`,
```

As before we start by establishing some basic lemmas and the existence of recursively defined functions:

```
  REPEAT GEN_TAC THEN REWRITE_TAC[GE; LE_0; real_gt] THEN STRIP_TAC THEN
  SUBGOAL_THEN `&0 <= Sigma` ASSUME_TAC THENL
   [FIRST_X_ASSUM(MP_TAC o SPEC `&1 / &2`) THEN REAL_ARITH_TAC;
    ALL_TAC] THEN
  SUBGOAL_THEN `!i. &0 < (B:num->real) i` ASSUME_TAC THENL
   [INDUCT_TAC THEN ASM_SIMP_TAC[REAL_LT_MUL; ADD1; REAL_LT_01]; ALL_TAC] THEN
  SUBGOAL_THEN `&0 < X` ASSUME_TAC THENL
   [ASM_REAL_ARITH_TAC; ALL_TAC] THEN
  SUBGOAL_THEN `&0 < sqrt X` ASSUME_TAC THENL
   [ASM_MESON_TAC[SQRT_POS_LT]; ALL_TAC] THEN
  MAP_EVERY ABBREV_TAC
   [`PSI:num->real#real->real = \i (u,t).
```

```
      if i = 0 then Sigma
      else Sigma + (&1 + Sigma) * t / (&2 * u * B i)`;
  `psi:num->real#real->real = \i (u,t).
      if i = 0 then sigma(X)
      else (&1 + sigma(X:real)) * (&1 - t / (&2 * u * B i)) - &1`] THEN
(X_CHOOSE_THEN `tau:num->real->real`
  (STRIP_ASSUME_TAC o REWRITE_RULE[ADD1]) o
 prove_recursive_functions_exist num_RECURSION)
  `(!u:real. tau 0 u = u) /\
   (!i u. tau (SUC i) u =
         beta (i + 1) * PSI i (u,tau i u) * tau i u + omega (i + 1))` THEN
(X_CHOOSE_THEN `Tl:num->real`
  (STRIP_ASSUME_TAC o REWRITE_RULE[ADD1]) o
 prove_recursive_functions_exist num_RECURSION)
  `Tl 0 = sqrt(X) /\
  !i. Tl (SUC i) = beta (i + 1) * Tl i - v (i + 1)` THEN
ABBREV_TAC
 `taup:num->real->real = \i u. (&1 + PSI i (u,tau i u)) * tau i u` THEN
```

and then instantiate the proxy theorem/corollary:

```
MP_TAC(ISPECL
 [`sqrt X`;
  `beta:num->real`;
  `omega:num->real`;
  `DSF:num->real->real`;
  `B:num->real`;
  `H:num->real`;
  `v:num->real`;
  `Tl:num->real`;
  `Tp:num->real`;
  `PSI:num->real#real->real`;
  `psi:num->real#real->real`;
  `tau:num->real->real`;
  `taup:num->real->real`]
  FULL_COROLLARY) THEN
REWRITE_TAC[GE; LE_0; real_gt] THEN ANTS_TAC THENL
```

The establishment of the hypotheses is now more complicated, mainly because of the more intricate proof that $R = \tilde{R}$.

```
 [REPEAT CONJ_TAC THENL
   [FIRST_X_ASSUM MATCH_ACCEPT_TAC;
    FIRST_X_ASSUM MATCH_ACCEPT_TAC;
    FIRST_X_ASSUM MATCH_ACCEPT_TAC;
    X_GEN_TAC `i:num` THEN MAP_EVERY EXPAND_TAC ["PSI"; "psi"] THEN
    REWRITE_TAC[] THEN ASM_CASES_TAC `i = 0` THEN ASM_SIMP_TAC[] THEN
    MATCH_MP_TAC(REAL_ARITH
     `abs x <= a /\ abs((&1 + x) * y) <= b
      ==> abs((&1 + x) * (&1 - y) - &1) <= a + b`) THEN
    ASM_SIMP_TAC[REAL_ABS_MUL] THEN
    MATCH_MP_TAC REAL_LE_MUL2 THEN REWRITE_TAC[REAL_ABS_POS] THEN
    ASM_SIMP_TAC[REAL_ARITH `abs x <= a ==> abs(&1 + x) <= &1 + a`] THEN
    REWRITE_TAC[REAL_ABS_DIV] THEN MATCH_MP_TAC REAL_EQ_IMP_LE THEN
    AP_TERM_TAC THEN
    MATCH_MP_TAC(REAL_ARITH `&0 < x ==> abs(&2 * x) = &2 * x`) THEN
    MATCH_MP_TAC REAL_LT_MUL THEN ASM_REWRITE_TAC[];
    MAP_EVERY X_GEN_TAC [`i:num`; `x:real`; `y:real`] THEN STRIP_TAC THEN
    EXPAND_TAC "PSI" THEN REWRITE_TAC[] THEN
    COND_CASES_TAC THEN ASM_REWRITE_TAC[REAL_LE_REFL; REAL_LE_LADD] THEN
    ASM_SIMP_TAC[REAL_LE_LADD; REAL_LE_LMUL_EQ; REAL_LE_DIV2_EQ;
                 REAL_ARITH `&0 <= s ==> &0 < &1 + s`; REAL_LT_MUL;
                 REAL_ARITH `&0 < &2 * x <=> &0 < x`] THEN
    REAL_ARITH_TAC;
    FIRST_X_ASSUM MATCH_ACCEPT_TAC;
    ASM_REWRITE_TAC[] THEN NO_TAC;
```

```
EXPAND_TAC "taup" THEN REWRITE_TAC[] THEN NO_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
X_GEN_TAC `i:num` THEN
FIRST_X_ASSUM(fun th -> GEN_REWRITE_TAC LAND_CONV [th]) THEN
EXPAND_TAC "psi" THEN REWRITE_TAC[] THEN
ASM_CASES_TAC `i = 0` THEN ASM_REWRITE_TAC[] THENL
 [ASM_SIMP_TAC[REAL_DIV_SQRT; REAL_LT_IMP_LE; REAL_ARITH
   `&2 * c / s * x / &2 = c * x / s`];
  ALL_TAC] THEN
REWRITE_TAC[REAL_MUL_LID; REAL_ARITH `&1 + x - &1 = x`] THEN
ASM_SIMP_TAC[] THEN REWRITE_TAC[real_div; GSYM REAL_MUL_ASSOC] THEN
AP_TERM_TAC THEN MATCH_MP_TAC(REAL_FIELD
 `&0 < b /\ &0 < s /\ r = (s - t / b / &2) * t
  ==> inv s * r = (&1 - t * inv(&2 * s * b)) * t`) THEN
ASM_REWRITE_TAC[] THEN
SUBGOAL_THEN `!j:num. Tl j / B j = sqrt X - H j`
ASSUME_TAC THENL
 [INDUCT_TAC THEN ASM_REWRITE_TAC[REAL_SUB_RZERO; REAL_DIV_1; ADD1] THEN
  UNDISCH_TAC `Tl(j:num) / B j = sqrt X - H j` THEN
  SUBGOAL_THEN `&0 < beta(j + 1) /\ &0 < B j` MP_TAC THENL
   [ASM_REWRITE_TAC[]; CONV_TAC REAL_FIELD];
  ASM_REWRITE_TAC[REAL_ARITH `s - (s - h) / &2 = (s + h) / &2`]] THEN
MATCH_MP_TAC(REAL_FIELD
 `!b. &0 < b /\ x / b = y / &2 * z / b ==> x = y / &2 * z`) THEN
EXISTS_TAC `(B:num->real) i` THEN ASM_REWRITE_TAC[REAL_ARITH
 `(x + h) / &2 * (x - h) = (x pow 2 - h pow 2) / &2`] THEN
ASM_SIMP_TAC[SQRT_POW_2; REAL_LT_IMP_LE] THEN
ASM_SIMP_TAC[REAL_EQ_LDIV_EQ] THEN
SPEC_TAC(`i:num`,`j:num`) THEN
MATCH_MP_TAC num_INDUCTION THEN CONJ_TAC THENL
 [ASM_REWRITE_TAC[] THEN REAL_ARITH_TAC; REWRITE_TAC[ADD1]] THEN
ONCE_REWRITE_TAC[ASSUME
`!i. R (i + 1) =
     beta (i + 1) * R i - v (i + 1) * (H (i + 1) + H i) / &2`] THEN
X_GEN_TAC `j:num` THEN SIMP_TAC[] THEN
REWRITE_TAC[ASSUME
 `!i. H (i + 1):real = H i + v (i + 1) / B (i + 1)`] THEN
REWRITE_TAC[ASSUME `!i. B (i + 1):real = beta (i + 1) * B i`] THEN
SUBGOAL_THEN `&0 < beta(j + 1) /\ &0 < B j` MP_TAC THENL
 [ASM_REWRITE_TAC[]; CONV_TAC REAL_FIELD];
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
FIRST_X_ASSUM MATCH_ACCEPT_TAC;
MAP_EVERY X_GEN_TAC [`i:num`; `p:real->real`] THEN DISCH_TAC THEN
EXPAND_TAC "PSI" THEN REWRITE_TAC[] THEN
ASM_CASES_TAC `i = 0` THEN ASM_SIMP_TAC[POSYNOMIAL_CONST] THEN
MATCH_MP_TAC POSYNOMIAL_ADD THEN
ASM_SIMP_TAC[POSYNOMIAL_CONST] THEN
MATCH_MP_TAC POSYNOMIAL_MUL THEN
ASM_SIMP_TAC[POSYNOMIAL_CONST; REAL_ARITH
 `&0 <= s ==> &0 <= &1 + s`] THEN
REWRITE_TAC[real_div; REAL_INV_MUL] THEN REWRITE_TAC[ REAL_ARITH
 `x * inv(&2) * inv y * z = (inv(&2) * z) * x / y`] THEN
MATCH_MP_TAC POSYNOMIAL_CMUL THEN
ASM_SIMP_TAC[REAL_LT_INV_EQ; REAL_ARITH
 `&0 < inv(&2) * x <=> &0 < x`] THEN
MATCH_MP_TAC POSYNOMIAL_VDIV THEN ASM_REWRITE_TAC[]];
```

The use of the result is very similar, however, and this quickly concludes the proof:

```
STRIP_TAC THEN EXISTS_TAC `tau:num->real->real` THEN
ASM_REWRITE_TAC[REAL_ADD_SUB] THEN CONJ_TAC THENL
 [EXPAND_TAC "PSI" THEN REWRITE_TAC[]; ALL_TAC] THEN
ASM_SIMP_TAC[REAL_ABS_DIV; REAL_LE_DIV2_EQ;
             REAL_ARITH `&0 < b ==> abs b = b`] THEN
```

```
  X_GEN_TAC `i:num` THEN
  FIRST_X_ASSUM(MP_TAC o SPECL [`&1 / &2`; `&1`]) THEN
  REWRITE_TAC[SUBSET; IN_REAL_INTERVAL; IN_ELIM_THM] THEN
  ANTS_TAC THENL [REAL_ARITH_TAC; ALL_TAC] THEN
  DISCH_THEN(MP_TAC o SPECL [`i:num`; `sqrt X`]) THEN
  ANTS_TAC THENL [ALL_TAC; ASM_MESON_TAC[REAL_LE_TRANS]] THEN
  CONJ_TAC THENL
   [MATCH_MP_TAC REAL_LE_RSQRT; MATCH_MP_TAC REAL_LE_LSQRT] THEN
  ASM_REAL_ARITH_TAC]);;
```

A.6 Automated instantiation (related to Table 1)

For convenience, we have implemented a HOL Light derived rule to instantiate the parameters of the theorems for division and square root and derive appropriately accurate error bounds for the successive approximations. A HOL Light derived rule is essentially a programmatic combination of more basic rules of inference, which is still doing full logical proof behind the scenes. Thus we can consider this as analogous to a spreadsheet producing results automatically as parameters are varied, but with the additional security of *proving* the result. We will not discuss the coding in detail, but it is very standard for such applications and can be understood by manually tracing through specific examples.

```
let BOUNDS_INSTATIATION =
  let pth = prove
     (`x <= a / b ==> &0 <= b ==> !a'. a <= a' ==> x <= a' / b`,
      REPEAT STRIP_TAC THEN TRANS_TAC REAL_LE_TRANS `a / b:real` THEN
      ASM_REWRITE_TAC[] THEN REWRITE_TAC[real_div] THEN
      MATCH_MP_TAC REAL_LE_RMUL THEN ASM_REWRITE_TAC[REAL_LE_INV_EQ]) in
  let rec calc rews (thb,ths) n =
    if n = 0 then [thb] else
    let oths = calc rews (thb,ths) (n - 1) in
    let th1 = CONV_RULE NUM_REDUCE_CONV (SPEC(mk_small_numeral(n - 1)) ths) in
    let th2 = GEN_REWRITE_RULE TOP_DEPTH_CONV (hd oths::rews) th1 in
    let th3 = CONV_RULE REAL_RAT_REDUCE_CONV th2 in
    th3::oths in
  fun th beta sigma omega n d ->
    let ith = BETA_RULE (SPECL [beta; sigma; omega] th) in
    let avs,itm = strip_forall(concl ith) in
    let hth = ASSUME (rand(lhand itm)) in
    let eth = MP (SPECL avs ith) (CONJ (REAL_ARITH(lhand(lhand itm))) hth) in
    let ev,ebod = dest_exists(concl eth) in
    let [th0;th1;bth] = CONJUNCTS(ASSUME ebod) in
    let (th_b,th_s) =
      let hths = CONJUNCTS hth in
      el (if th = BOUND_THEOREM_DIV then 6 else 4) hths,
      el (if th = BOUND_THEOREM_DIV then 11 else 9) hths in
    let bths = calc [] (th_b,th_s) n in
    let tths_lo =
      calc bths (SPEC (if th = BOUND_THEOREM_DIV then `&1 / &4` else `&1 / &2`)
                 th0,
            SPEC (if th = BOUND_THEOREM_DIV then `&1 / &4` else `&1 / &2`)
                 (GEN_REWRITE_RULE I [SWAP_FORALL_THM] th1)) n
    and tths_hi =
      calc bths (SPEC `&1:real` th0,
            SPEC `&1:real` (GEN_REWRITE_RULE I [SWAP_FORALL_THM] th1)) n in
    let aths = map
     (CONV_RULE REAL_RAT_REDUCE_CONV o
      REWRITE_RULE(tths_lo@tths_hi) o
      C SPEC bth o mk_small_numeral) (0--n) in
    let weaken th =
      let th1 = MATCH_MP pth th in
      let th2 = GEN_REWRITE_CONV RAND_CONV bths (lhand(concl th1)) in
```

```
    let th3 = CONV_RULE(RAND_CONV REAL_RAT_REDUCE_CONV) th2 in
    let th4 = MP th1 (EQT_ELIM th3) in
    let rr = rat_of_term(lhand(lhand(snd(dest_forall(concl th4))))) in
    let yy = pow10 d in
    let xx = ceiling_num(yy */ rr) in
    let th5 = SPECL [mk_numeral xx; mk_numeral yy] DECIMAL in
    let th6 = SPEC (lhand(concl th5)) th4 in
    MP th6 (EQT_ELIM(REAL_RAT_REDUCE_CONV(lhand(concl th6)))) in
  let ath = end_itlist CONJ (map weaken aths) in
  GENL avs (DISCH_ALL (CHOOSE(ev,eth) ath));;
```

The toplevel function takes a number of parameters

- `th` is the bounds theorem to instantiate, which will be `BOUND_THEOREM_DIV` or `BOUND_THEOREM_SQRT`.
- `beta`, `sigma` and `omega` are HOL term instantiations for the particular values of β, Σ and Ω.
- `n` is the number of iterations for which bounds are desired: an input of n will result in bounds for $H_0, H_1, \ldots, H_n$.
- `d` is the number of fractional digits in the decimal representation of the digit bounds.

For example the instantiation:

```
BOUNDS_INSTATIATION BOUND_THEOREM_SQRT
 `(\i. if i = 2 then &32 else if i = 5 then &64 else &128):num->real`
 `inv(&2 pow 8):real`
 `(\i. if i = 0 then &1 / &2 else &9 / &16):num->real`
 7 6;;
```

results automatically in the following theorem giving bounds to 6 places after the decimal point for the iterations $H_0, \ldots, H_7$ for the square root algorithm with (somewhat arbitrary) parameters:

```
  |- !B DSF H R Tp X g sigma v.
         &1 / &4 <= X /\
         X < &1 /\
         (!x. &1 / &4 <= x /\ x < &1
              ==> g x = (&1 + sigma x) / sqrt x /\
                  abs (sigma x) <= inv (&2 pow 8)) /\
         (!i. i >= 1
              ==> (!x. abs (x - DSF i x) <=
                       (if i = 0 then &1 / &2 else &9 / &16))) /\
         B 0 = &1 /\
         H 0 = &0 /\
         R 0 = X / &2 /\
         (!i. Tp i = (if i = 0 then &2 else &1) * g X * R i) /\
         (!i. v (i + 1) =
              DSF (i + 1)
              ((if i + 1 = 2 then &32 else if i + 1 = 5 then &64 else &128) *
               Tp i)) /\
         (!i. B (i + 1) =
              (if i + 1 = 2 then &32 else if i + 1 = 5 then &64 else &128) *
              B i) /\
         (!i. H (i + 1) = H i + v (i + 1) / B (i + 1)) /\
         (!i. R (i + 1) =
              (if i + 1 = 2 then &32 else if i + 1 = 5 then &64 else &128) *
              R i -
```

```
     v (i + 1) * (H (i + 1) + H i) / &2)
==> abs (sqrt X - H 0) <= #1.000000 / B 0 /\
    abs (sqrt X - H 1) <= #1.062500 / B 1 /\
    abs (sqrt X - H 2) <= #0.836978 / B 2 /\
    abs (sqrt X - H 3) <= #0.998973 / B 3 /\
    abs (sqrt X - H 4) <= #1.062231 / B 4 /\
    abs (sqrt X - H 5) <= #0.828059 / B 5 /\
    abs (sqrt X - H 6) <= #0.976530 / B 6 /\
    abs (sqrt X - H 7) <= #1.050765 / B 7
```

Using similar simple invocations we can exactly check the main bounds given in Table 1. Where they differ in the last digit, the difference arises because our theorems are returning actual bounds whereas the table just rounds the bounds to nearest.

References

[1] Alexander Barvinok. *A Course in Convexity*, volume 54 of *Graduate Texts in Mathematics*. American Mathematical Society, 2002.

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[3] W S Briggs and David W Matula. A 17× 69 bit multiply and add unit with redundant binary feedback and single cycle latency. In *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 163–170. IEEE, 1993.

[4] M Daumas and D Matula. Recoders for partial compression and rounding. *Ecole Normale Supérieure de Lyon, Research Report LIP-RR1997-01*, 1997.

[5] Marc Daumas and David W Matula. Further reducing the redundancy of a notation over a minimally redundant digit set. *The Journal of VLSI Signal Processing*, 33(1):7–18, 2003.

[6] Richard J Dufin, Elmor L Peterson, and Clarence Zener. *Geometric programming-theory and application*. John Wiley, 1967.

[7] Milos D Ercegovac and Tomas Lang. On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, 36(7):895–897, 1987.

[8] Milos D Ercegovac and Tomas Lang. On-the-fly rounding for division and square root. In *Proceedings of 9th IEEE Symposium on Computer Arithmetic*, pages 169–173. IEEE, 1989.

[9] Milos D Ercegovac and Tomas Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[10] Milos D Ercegovac and Tomás Lang. On recoding in arithmetic algorithms. In *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 531–535. IEEE, 1994.

[11] Milos D Ercegovac, Tomas Lang, and Paolo Montuschi. Very-high radix division with prescaling and selection by rounding. *IEEE Transactions on Computers*, 43(8):909–918, 1994.

[12] John Harrison. HOL Light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FM-CAD96)*, volume 1166 of Lecture Notes in Computer Science, pages 265–269. Springer, 1996.

[13] John Harrison. The HOL Light tutorial. Unpublished manual available at `http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial.pdf`, 2006.

[14] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, third edition, November 1997.

[15] Tomas Lang and Paolo Montuschi. Very-high radix combined division and square root with prescaling and selection by rounding. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 124–131. IEEE, 1995.

[16] Constantin Niculescu and Lars-Erik Persson. *Convex Functions and their Applications: A Contemporary Approach*. Springer Science & Business Media, 2006.

[17] Roger Webster. *Convexity*. Oxford University Press, 1995.