An awkward first method for computing $\sqrt{x}$ using only $+$ and $*$

Suppose we seek the square root of A on a machine that can only do addition and multiplication. We assume $A \geq 0$ for simplicity, and of course if $A = 0$ then $\sqrt{A} = 0$ and we are done. Otherwise the computer has something like this way of representing x as bits:[1]

$$A = 1.01101001_2 \times 2^b.$$

There are bits representing the binary "mantissa" 1.01101001_2, in base 2. As in scientific notation, the mantissa has the decimal point in the same place for all A. Bits also represent the power b, which is an integer.

If b is even then we know the exact square root of 2^b, but not so if b is odd. (Consider the $b = 1$ case. I do not know $\sqrt{2}$ exactly.) But we can shift the decimal point so that the power is even. For example, we can write $A = 10.1101001_2 \times 2^{b-1}$ if b above is odd. Thus, in any case, we can write

$$A = x \times 2^{2k}$$

for x in a range $1 = 1.0000000_2 \leq x \leq 11.111111\ldots_2 = 3.99999\cdots < 4$. Here k is an integer. Then

$$\sqrt{A} = \sqrt{x} \times 2^k.$$

This is why we need a good approximation for $\sqrt{x}$ where $x \in [1, 4]$. With such a good approximation we can approximate $\sqrt{A}$ for A of any magnitude.

The tool we have right now is Taylor's Theorem. Let $f(x) = \sqrt{x}$. Let us try a base point $x_0 = 4$. This is a good choice because the interval of convergence of the Taylor series will be stopped by $x = 0$ but it should include $x \in [1, 4]$. Equally important, the square root of 4 is, after all, known! Now start computing derivatives as usual:

$$\begin{aligned}
f(x) &= x^{1/2} & f(x_0) &= 2 \\
f'(x) &= \frac{1}{2}x^{-1/2} & f'(x_0) &= \frac{1}{2 \cdot 2} \\
f''(x) &= \frac{-1}{2^2}x^{-3/2} & f''(x_0) &= \frac{-1}{2^2 \cdot 2^3} \\
f'''(x) &= \frac{+3}{2^3}x^{-5/2} & f''(x_0) &= \frac{+3}{2^3 \cdot 2^5} \\
f^{(4)}(x) &= \frac{-3 \cdot 5}{2^4}x^{-7/2} & f^{(4)}(x_0) &= \frac{-3 \cdot 5}{2^4 \cdot 2^7} \\
&\vdots & &\vdots \\
f^{(k)}(x) &= \frac{(-1)^{k-1}3 \cdot 5 \cdot 7 \cdot \cdots \cdot (2k-3)}{2^k}x^{-(2k-1)/2} & f^{(k)}(x_0) &= \frac{(-1)^{k-1}3 \cdot 5 \cdot 7 \cdot \cdots \cdot (2k-3)}{2^k \cdot 2^{2k-1}}
\end{aligned}$$

[1] Section 1.3 of the textbook (J. Epperson, *An Introduction to Numerical Methods and Analysis*, rev. ed., Wiley, 2007) has a more authentic form, but nothing is fundamentally wrong about the story here.

Because we want lots of accuracy, let us plan to find n so that the Taylor polynomial is within 10^{-16} for all $x \in [1,4]$:

$$|f(x) - p_n(x)| = |R_n(x)| \leq 10^{-16}.$$

But

$$\begin{aligned} |R_n(x)| &= \frac{|f^{(n+1)}(\xi)|}{(n+1)!}|x-x_0|^{n+1} = \frac{\frac{3\cdot 5\cdot 7\cdot\cdots\cdot(2n-1)}{2^{n+1}}\xi^{-(2n+1)/2}}{(n+1)!}|x-4|^{n+1} \\ &= \frac{(3\cdot 5\cdot 7\cdot\cdots\cdot(2n-1))\,|x-4|^{n+1}}{2^{n+1}\,(n+1)!\,\xi^{(2n+1)/2}} \end{aligned}$$

from the derivative expressions, using $k = n+1$. As usual, all we know about ξ is that it is between $x_0 = 4$ and x. Thus both $x \in [1,4]$ and $\xi \in [1,4]$. Considering the worst cases and simplifying the factorial expressions,

$$|R_n(x)| \leq \frac{(3\cdot 5\cdot 7\cdot\cdots\cdot(2n-1))\,3^{n+1}}{2^{n+1}\,(n+1)!\,1^{(2n+1)/2}} = \frac{(2n-1)!\,3^{n+1}}{2^{n+1}\,2^{n-1}\,(n-1)!\,(n+1)!} = \frac{(2n-1)!\,3^{n+1}}{2^{2n}\,(n-1)!\,(n+1)!}.$$

Unfortunately, as shown in class the right-hand estimate in the above *grows* exponentially, as shown in Figure 1.

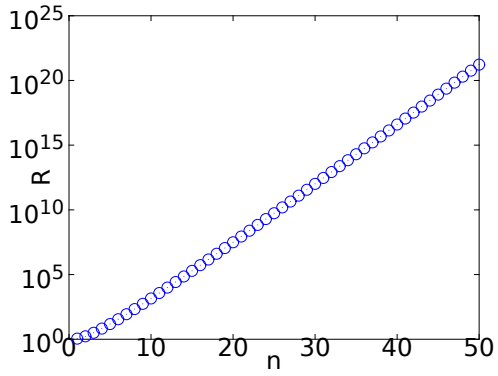

FIGURE 1. Estimate $\frac{(2n-1)!\,3^{n+1}}{2^{2n}\,(n-1)!\,(n+1)!}$ grows exponentially with n.

What is going on? One possibility is that $|R_n(x)|$ does not decrease with n, and thus that we will not be able to find n so that $|R_n(x)| \leq 10^{-16}$ for $x \in [1,4]$. But another possibility is that the worst-case estimate above is too pessimistic.

To diagnose let us plot $f(x) = \sqrt{x}$ and some polynomials $p_n(x)$ as well. In fact, we can thereby *see* the remainder term $R_n(x)$, as it is the difference $R_n(x) = f(x) - p_n(x)$. The polynomials have these formulas,

$$\begin{aligned} p_n(x) &= \sum_{k=0}^{n} \frac{f^{(k)}(4)}{k!}(x-4)^k \\ &= 2 + \frac{1}{4}(x-4) - \frac{1}{2^6}(x-4)^2 + \sum_{k=3}^{n} \frac{(-1)^{k-1}3\cdot 5\cdot 7\cdot\cdots\cdot(2k-3)}{2^k\cdot 2^{2k-1}\cdot k!}(x-4)^k \\ &= 2 + \frac{1}{4}(x-4) - \frac{1}{2^6}(x-4)^2 + \sum_{k=3}^{n} \frac{(-1)^{k-1}(2k-3)!}{2^{4k-3}\cdot(k-2)!\cdot k!}(x-4)^k \end{aligned}$$

(I have written out $1, x, x^2$ terms explicitly because it takes a while for the pattern to get "established".)

Sums like the above are, essentially, *programs*. By being specific enough to use summation notation, we have basically written a code. Here is a MATLAB/OCTAVE code which generates Figure 2.

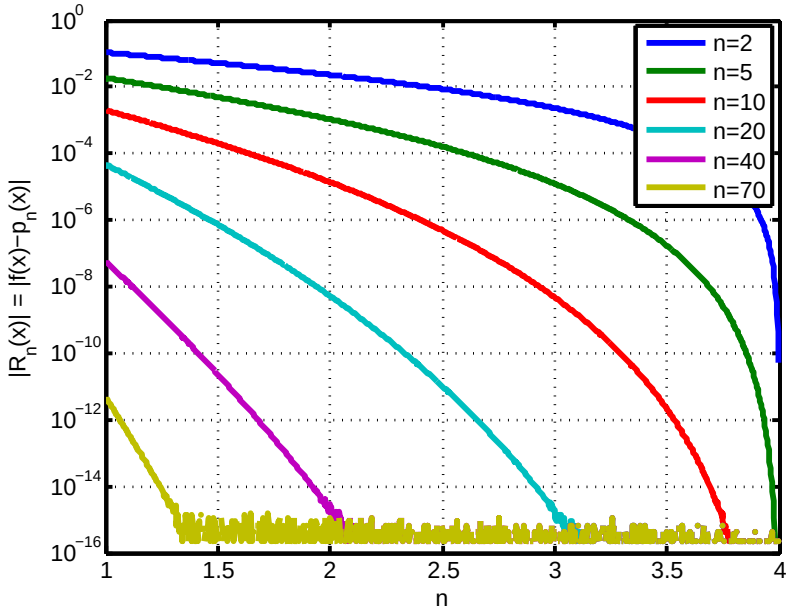

FIGURE 2. Plots of the error $|R_n(x)| = |f(x) - p_n(x)|$. Note that $p_{70}(x)$ gives an error which is less than 10^{-11} on the whole interval $[1, 4]$.

plotpolysqrt.m

```
% PLOTPOLYSQRT  Show how well Taylor polynomials p_n(x) approximate
%               the square root function on the interval [1,4].

x = linspace(1,4,1001)';              % points on [1,4], as a column
f = sqrt(x);
p1 = 2 + (1/4)*(x-4);                 % get started explicitly
p2 = p1 - (1/64)*(x-4).^2;

N = 70;                               % this many p_n(x)
pn = zeros(length(x),N);              % space for all p_n(x)
pn(:,1) = p1;
pn(:,2) = p2;
sign = -1;
for n = 3:N                           % count through the n-values
  sign = (-1) * sign;                 % generate alternating sign
  coeff = factorial(2*n-3) ...
            / ( 2^(4*n-3) * factorial(n-2) * factorial(n) );
  pn(:,n) = pn(:,n-1) + sign * coeff * (x-4).^n; % add next term
end
```

```
for n=1:N                                  % replace each polynomial
  pn(:,n) = abs(f - pn(:,n));              %    with its error
end

semilogy(x,pn(:,[2 5 10 20 40 70])), grid on  % log scale on y
legend('n=2','n=5','n=10','n=20','n=40','n=70')
xlabel n, ylabel('|R_n(x)| = |f(x)-p_n(x)|')
```

Figure 2 shows that we got pretty good accuracy, with $|R_n(x)| < 10^{-11}$ for $n = 70$ on $[1, 4]$, though not 10^{-16}. Thus the problem with the analysis above was not that the remainder $R_n(x)$ itself was growing in magnitude as n increased, but that our *estimate* of the remainder was growning like that. For our purpose it is *not* good enough as an estimate.

Thus we can write an admittedly awkward code for $\sqrt{x}$ that uses stored coefficients of $p_{70}(x)$. It starts by extracting the mantissa and power the input number x using the "`log2`" function which is designed for this purpose, but this could easily be done in hardware by moving bits left and right. Similarly, at the end it uses exponential to replace "2^b" with "$2^{b/2}$", but this is a bit manipulation that can be done in hardware.

edsqrt.m

```
function y = edsqrt(x)
% EDSQRT  An awkward Taylor polynomial method to compute sqrt(x)
%         using only + and *.

if x < 0, error('EDSQRT(x) only works for x >= 0'), end
if x == 0, y = 0; return, end          % done with x = 0 case

% extract mantissa and exponent in base 2 scientific notation;
%   (in a "real" implementation this would be done by hardware)
[mant,expo] = log2(x);
mant = mant * 2;                       % adjust to match in-class:
expo = expo - 1;                       %    mantissa is in [1,2]
if mod(expo,2)==1                      % is exponent odd?
  expo = expo - 1;                     % fix exponent to be even
  mant = mant * 2;
end
x = mant;                              % now 1 <= x <= 4

% stored coefficients; computed by a PLOTPOLYSQRT run
c = [2.000000000000e+00  2.500000000000e-01  -1.562500000000e-02 ...
  1.953125000000e-03  -3.051757812500e-04  5.340576171875e-05 ...
  -1.001358032227e-05  1.966953277588e-06  -3.995373845100e-07   ...
  8.323695510626e-08  -1.768785296008e-08  3.818968252745e-09   ...
  -8.353993052879e-10  1.847517694387e-10  -4.123923424970e-11   ...
  9.278827706183e-12  -2.102234402182e-12  4.791857828503e-13   ...
  -1.098134085699e-13  2.528598223648e-14  -5.847383392186e-15   ...
  1.357428287472e-15  -3.162190896951e-16  7.389902639615e-17   ...
  -1.732008431160e-17  4.070219813225e-18  -9.588498598464e-19   ...
```

```
  2.263951057971e-19  -5.356669913948e-20  1.269900195117e-20   ...
  -3.016012963403e-21  7.175192130676e-22  -1.709713749888e-22   ...
  4.079998721323e-23  -9.749996944338e-24  2.333034983110e-24   ...
  -5.589562980367e-25  1.340739768939e-25  -3.219539576729e-26   ...
  7.739277828676e-27  -1.862263727525e-27  4.485330319344e-28   ...
  -1.081284987699e-28  2.608914359855e-29  -6.299935243968e-30   ...
  1.522484350626e-30  -3.682095304502e-31  8.911454061429e-32   ...
  -2.158242780502e-32  5.230435309891e-33  -1.268380562649e-33   ...
  3.077688129956e-34  -7.472271661672e-35  1.815198068755e-35   ...
  -4.411939750446e-36  1.072903530222e-36  -2.610412607014e-37   ...
  6.354293846021e-38  -1.547489665087e-38  3.770366768750e-39   ...
  -9.190268998828e-40  2.241069694386e-40  -5.467125665741e-41   ...
  1.334239001758e-41  -3.257419437886e-42  7.955620550223e-43   ...
  -1.943702748066e-43  4.750467537250e-44  -1.161419453041e-44   ...
  2.840428010154e-45  -6.948904239127e-46];
y = c(1);
for n = 1:70
  y = y + c(n+1) * (x-4)^n;            % "^n" could be multiplication
end

y = y * 2^(expo/2);                    % divide exponent by 2 to sqrt
```

So, does it work? Does it get good accuracy? Does it handle big and small numbers?

Here is how it compares to the built-in "`sqrt`" function. Note that I ask MATLAB/OCTAVE to show more digits. Also, `edsqrt.m` appropriately complains about negative inputs.

```
>> format long
>> x=3.99; edsqrt(x), sqrt(x)
ans =     1.99749843554382
ans =     1.99749843554382
>> x=3141592600; edsqrt(x), sqrt(x)
ans =     56049.9116859251
ans =     56049.9116859251
>> x=0.000000000071828182845; edsqrt(x), sqrt(x)
ans = 2.68007803701683e-05
ans = 2.68007803701683e-05
>> x=-10; edsqrt(x), sqrt(x)
error: EDSQRT(x) only works for x >= 0
...
```

So far so good. In fact we are getting 15 digit agreement with the built -in function for numbers which are very small and very big; our "bitwise" manipulation seems to work to put the Taylor problem onto the interval $[1, 4]$.

But there is a small problem, which we could predict from Figure 2. Namely when we look at x near 1 in the interval $[1, 4]$, there is reduced accuracy. It goes away by about $x = 1.5$:

```
>> x=1.0; edsqrt(x), sqrt(x)
ans =     1.00000000000481
ans =     1.00000000000000
>> x=1.1; edsqrt(x), sqrt(x)
ans =     1.04880884817055
ans =     1.04880884817015
>> x=1.5; edsqrt(x), sqrt(x)
ans =     1.22474487139159
ans =     1.22474487139159
```

As we will see there are shorter, faster ways to get accurate square root computations using only $+$ and $*$. We will return to this problem!