1 RECIPROCAL & SQUARE ROOT

1

1.1 INTRODUCTION

Many early computers had neither a hardware division x/y nor a reciprocation $1/x$ operation. Needless to say, these computers did not have hardware $\sqrt{x}$ nor the transcendental functions such as $\sin(x), \cos(x)$, etc. These operations had to be implemented in software using the hardware's addition and multiplication operations.

Today's processors (Intel, AMD, etc.) have many built-in operations. These include the standard arithmetic operations along with $\sqrt{x}$, $\log_e x$, and the trigonometric functions. A closer look at these processors shows that the only true hardware operations are addition and multiplication.

Here is an excerpt from the *Intel Application Note* on "Divide, Square Root, and Remainder Algorithms" (Doc No. 248725-002)

> The IA-64 architecture specifies two approximation instructions, **frcpa** and **frsqrta**, that are designed to support efficient and IEEE-correct software implementations of division, square root and remainder. Deferring most of the division and square root computations to software offers several advantages. Most importantly, since each operation is broken down into several simpler instructions, these individual instructions can be scheduled more flexibly in conjunction with the rest of the code, increasing the potential for parallelism. In particular:
>
> - Since the underlying operations are fully pipelined, the division and square root operations inherit the pipelining, allowing high throughput.
> - If a perfectly rounded IEEE-correct result is not required (e.g. in graphics applications), faster algorithms can be substituted.
>
> Intel provides a number of recommended division and square root algorithms, in the form of short sequences of straight-line code written in assembly language for the IA-64 architecture. The intention is that these can be inlined by compilers, used as the core of mathematical libraries, or called on as macros by assembly language programmers. It is expected that these algorithms will

serve all the needs of typical users, who when using a high-level language might be unaware of how division and square root are actually implemented.

⋮

All the algorithms are implemented as segments of straight-line code, which perform an initial approximation step (**frcpa** or **frsqrta**) and then refine the resulting approximation to give a correctly rounded result, using power series or iterative methods such as the Newton-Raphson or Goldschmidt iteration.

This essay attempts to explain the high-level details of the reciprocal and square root algorithms, which use addition and multiplication only.

1.2 NEWTON'S METHOD FOR THE RECIPROCAL

We may define the reciprocal of some number $a > 0$ as a zero of the function $f(x) = 1/x - a$. Using this function in Newton's method

$$\boxed{x_{k+1} := x_k - \frac{f(x_k)}{f'(x_k)}} \tag{1.1}$$

gives

$$x_{k+1} := x_k - \frac{(1/x_k - a)}{(-1/x_k^2)} = x_k + (x_k - a x_k^2)$$

Thus we get the Newton iteration function for reciprocals :

$$\boxed{x_{k+1} := x_k(2 - a x_k) \triangleq T(x_k).} \tag{1.2}$$

Notice that this iteration function uses multiplication and subtraction only.

The steps of Newton's method are shown graphically in Figure 1.

1.2.1 Analysis of Newton's Reciprocal Method

Newton's method (1.1 and 1.2) above are examples of a *fixed point iteration function.*[2] Under conditions to be derived below, the sequence generated by $x_{k+1} := T(x_k)$ converges to a fixed point $x = T(x)$.

The first question to answer is this : is $1/a$ a fixed point of $T(x)$ above? At a fixed point of T we have $x = T(x)$, or $x = x(2 - ax)$. The equation $x = 2x - ax^2$ has two roots, $x = 0$ and $x = 1/a$. These are the fixed points of $T(x)$ because they satisfy the fixed point equation. Whether or not the sequence $x_{k+1} := T(x_k)$ converges to one or other or both of them is the next question.

Under what condition(s) will the iterative method $x_{k+1} := x_k(2 - a x_k) = T(x_k)$, converge to $1/a$? It is obvious, by looking at the graph above, that the sequence, starting at x_0, converges when $0 < x_0 \le 1/a$. We now determine the domain of convergence in a more rigourous manner.

[2] This is also called *successive approximation.*

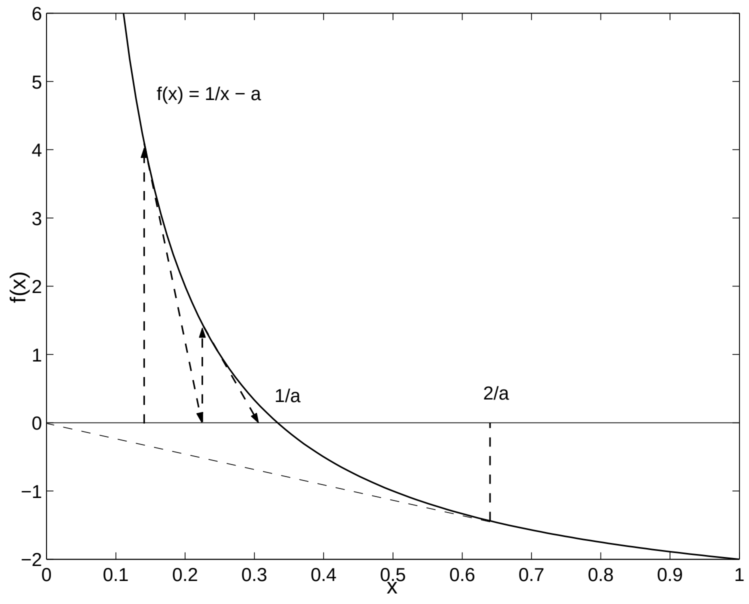

Figure 1.1 : Newton's Method for the Reciprocal of a

We use a method from the book by Demidovitch & Maron who express the error at the kth iteration in terms of the initial error. This is the most direct and useful approach and uses no theory other than the convergence of the geometric series.

Let $e_k = |x_k - 1/a| / 1/a = 1 - ax_k$ be the relative error at the kth iteration of $x_{k+1} := x_k(2 - ax_k)$. Then we have

$$\begin{aligned} e_k = 1 - ax_k &= 1 - ax_{k-1}(2 - ax_{k-1}) \\ &= 1 - 2x_{k-1} + ax_{k-1}^2 \\ &= (1 - ax_{k-1})^2 = e_{k-1}^2 \end{aligned} \tag{1.3}$$

This result, $e_k = e_{k-1}^2$, shows immediately that the sequence has 2nd-order convergence, which we normally expect of Newton's method. Using successive substitution we get

$$\begin{aligned} e_k &= e_{k-1}^2 = (e_{k-2}^2)^2 = e_{k-2}^{2^2} = (e_{k-3}^{2^2})^2 \\ &= e_{k-3}^{2^3} = \cdots = e_0^{2^k}, \text{ and we get} \end{aligned}$$

$$\boxed{e_k = e_0^{2^k} = (1 - ax_0)^{2^k}.} \tag{1.4}$$

A necessary and sufficient condition for the sequence $\{e_k\}$ to converge is $|ax_0 - 1| < 1$. This condition gives $|ax_0 - 1| < 1 \Rightarrow -1 < ax_0 - 1 < 1 \Rightarrow 0 < ax_0 < 2$, and we get the domain of convergence

$$\boxed{0 < x_0 < \frac{2}{a}} \qquad \text{Domain of Convergence} \tag{1.5}$$

We note that this interval is twice the width of the interval obtained by inspection.

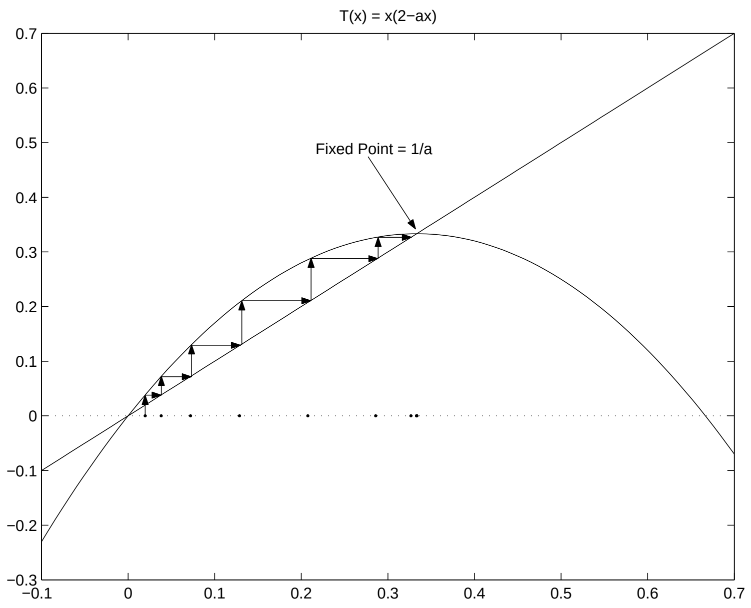

Figure 1.2 : Converging to a fixed point of $T(x) = x_k(2 - ax_k)$

Starting Accuracy

The number of iterations needed to attain full precision accuracy depends on, (i) the order of convergence of the algorithm, (ii) the number of digits of precision in the floating point system, and (iii) the accuracy of the starting point x_0.

Newton's method for the reciprocal has 2nd-order convergence, as shown in equations (1.3) and (1.4) above. This means that the number of accurate digits doubles at each iteration, *if the initial relative error* $e_0 < 1$. If we are using IEEE double-precision floating point arithmetic then we have $p = 53$ bits precision. If x_0 has 1 bit accuracy, i.e., $e_0 = 2^{-1}$, then we have 2 bit accuracy after the first iteration, 4 bits, 8 bits, 16 bits, 32 bits, and 64 bits accuracy after iterations 2,3,4,5, and 6. Obviously the higher the starting accuracy, the lower the number of iterations needed to reach full-precision accuracy.

In general, with $p-$digit base b arithmetic and order of convergence m, full-precision accuracy is attained after n iterations when $e_0^{m^n} = b^{-p}$. Taking $e_0 = b^{-s}$ we get $m^n = p/s$ and so we need, for $m \geq 2$,

$$n = \frac{\log_b p - \log_b s}{\log_b m} \quad \text{iterations for full-precision accuracy.} \tag{1.6}$$

Although this formula should not be taken too literally, it does show how precision p, starting accuracy s, and order of convergence m, interact.

We will examine this question in more detail when we implement the reciprocal algorithm.

1.2.2 Using the Reciprocal Algorithm Symbolically

It is instructive to see if this reciprocal algorithm will work symbolically. The iteration function is

$$x_{k+1} = x_k \times (2 - a \times x_k) \triangleq T(x_k).$$

We wish to generate symbolically the sequence defined by $\{x_{k+1} := T(x_k)\}$. We begin by defining the iteration function with the Maxima command : [3]

%i20 T(x) := x*(2-a*x);

%o20 $\mathrm{T}(x) := x\,(2 - a\,x)$

%i21 [r:T(1/(2*a)),float(r)]; We start with the initial value $1/2a < 1/a$.

`r:T(r)` is Maxima's odd way of storing $T(r)$ in r.

%o21 $\left[\frac{3}{4}\frac{1}{a}, 0.75\,\frac{1}{a}\right]$

%i22 [r:T(r), float(r)]; This applies $T()$ to the previous output.

%o22 $\left[\frac{15}{16}\frac{1}{a}, 0.9375\,\frac{1}{a}\right]$

%i23 [r:T(r), float(r)];

%o23 $\left[\frac{255}{256}\frac{1}{a}, 0.99609375\,\frac{1}{a}\right]$

%i24 [r:T(r), float(r)];

%o24 $\left[\frac{65535}{65536}\frac{1}{a}, 0.99998474121094\,\frac{1}{a}\right]$

%i25 [r:T(r), float(r)];

%o25 $\left[\frac{4294967295}{4294967296}\frac{1}{a}, 0.99999999976717\,\frac{1}{a}\right]$

%i26 [r:T(r), float(r)];

%o26 $\left[\frac{18446744073709551615}{18446744073709551616}\frac{1}{a}, 1.0\frac{1}{a}\right]$ Converged in 6 iterations with 16-digit precision

This clearly shows the second-order convergence of the sequence to $1/a$, with the number of correct digits doubling at each iteration.

Starting at a value $7/4a > 1/a$ gives the following

%i100 T(x) := x*(2-a*x);

%o100 $\mathrm{T}(x) := x\,(2 - a\,x)$

%i101 [r:T(7/(4*a)),float(r)];

%o101 $\left[\frac{7}{16}\frac{1}{a}, \frac{0.4375}{a}\right]$

%i102 [r:T(r), float(r)];

%o102 $\left[\frac{175}{256}\frac{1}{a}, \frac{0.68359375}{a}\right]$

%i103 [r:T(r), float(r)];

%o103 $\left[\frac{58975}{65536}\frac{1}{a}, \frac{0.89988708496094}{a}\right]$

%i104 [r:T(r), float(r)];

%o104 $\left[\frac{4251920575}{4294967296}\frac{1}{a}, \frac{0.98997740424238}{a}\right]$

%i105 [r:T(r), float(r)];

%o105 $\left[\frac{18444891053520699775}{18446744073709551616}\frac{1}{a}, \frac{0.99989954757428}{a}\right]$

%i106 [r:T(r), float(r)];

%o106 $\left[\frac{340282363487254643170862122773919122175}{340282366920938463463374607431768211456}\frac{1}{a}, \frac{0.99999998990931}{a}\right]$

%i107 [r:T(r), float(r)];

%o107 $\left[\frac{115792089237316183633386407270104736332397123253121898361245991732072020542975}{115792089237316195423570985008687907853269984665640564039457584007913129639936}\frac{1}{a}, \frac{1.0}{a}\right]$

[3]lines beginning with `%i` n are input, and those with `%o` n are output.

We can see that the sequence always converges monotonely to $1/a$ from below.
Let us choose a starting value $x_0 = 3/a$, which is outside the domain of convergence $0 \leq x_0 \leq 2/a$.

i5 T(3/a);

o5 -3 $\frac{}{a}$

i6 T(

o6 -15$\frac{}{a}$

i7 T(

o7 -255$\frac{}{a}$

i8 T(

o8 -65535$\frac{}{a}$

This clearly shows the sequence diverging. The other fixed point of T is $x = 0$, which can be reached only from the starting point $x_0 = 2/a$. It then converges to 0 in one step (see Figure 2 above).

A Note on the Starting Value. The starting values for x_0 above were $1/2a$, $7/4a$, and $3/a$. This may seem to be 'begging the question' because we include $1/a$ in the starting value. Note however, that this is merely a symbol and it is not evaluated. Only the numerical parts of the function $T(x)$ are evaluated. We call such calculations **semi-symbolic**. We could call them semi-numeric but this would confuse them with the semi-numeric algorithms in Knuth's *Seminumerical Algorithms* **??**.

1.2.3 Implementation, Analysis, and Testing of the Reciprocal Algorithm

The reciprocal algorithm was implemented in various ways using MATLAB. The first, Recip1(a,x0), requires the user to supply a starting value x_0. This is useful for seeing the effects of good and bad starting values. These are shown in Table I. I the second implementation, Recip2(a), we assume that e and f exist such that $a = 2^e f, \quad 1/2 \le f < 1$. This is true in binary arithmetic. If we calculate e and f then $1/a = 2^{-e}(1/f)$. We use $x_0 = 2^{-e}$ as the starting value.

```
%=====================================  | %====================================
% Recip1 is a naive implementation      | % Recip2 uses argument reduction to
% of 1/a                                | % obtain an estimate of 1/a
  function xnew = Recip1(a,x0);         |    function xnew = Recip2(a);
                                        |
  xold = x0;                            |    if a == 1
  for k = 1:1000                        |       xnew = a;
      xnew = xold*(2 - a*xold);         |       return ;
      if abs(xnew - xold) <= eps*xnew   |    end;
      return                            |    f = a;
  end;                                  |    e = 0;
  xold = xnew;                          |    while f >= 1
                                        |       e = e + 1;
                                        |       f = f*0.5;
                                        |    end;
                                        |    while f < 1/2
                                        |       e = e - 1;
                                        |       f = f*2;
                                        |    end;                %  1/2 <= f < 1
                                        |    xold = 2^(-e);
                                        |    relerr = 1.0;
                                        |    while relerr > eps
                                        |       xnew = xold*(2 - a*xold);
                                        |       relerr = abs(xnew-xold)/xnew
                                        |       xold = xnew;
                                        |    end;
                                        |
                                        | %---------------------------
```

We see in the Recip2 above that one or other, but not both of the while-loops reduces or increases f until we have $1/2 \le f < 1$, where e counts the number of multiplications or divisions by 2 needed to reduce f. We then use $x_0 = 2^{-e}$ as the starting value.

Recip3 is the final implementation of the reciprocal algorithm. It is just a slick version of Recip2. There are three points that deserve special attention :

1. It is more robust because it checks for special cases $a = \infty$ and $a = 0$. It does this at the start because these values might cause difficulties in the approximation loop. It does not check for negative arguments. Perhaps it should.

2. It uses the MATLAB log 2(a) function to determine f and e such that $2^e f = a$, with $1/2 \le f < 1$.

3. It uses the correction form of the iteration formula : $e_k = 1 - ax_k$ and $x_{k+1} := x_k + e_k x_k$, in keeping with the advice given in Chapter 4. This 2-part formula is mathematically identical to the formula in equation (2) above. It has the benefit that e_k is in fact the relative error in the estimate x_k of $1/a$.[4] This means that we do not have to make a separate calculation of it for convergence checking.

[4] $e_k = |1/a - x_k|/|1/a| = 1 - ax_k$. Note in this implementation that $x_k < 1/a$, always.

```
%=============================================================
% Recip3 uses argument reduction to obtain an estimate of 1/a.
% Find f and e such that a = f*2^e and 1/2 <= f < 1
% Use 2^(-e) as the initial value.
% Derek O'Connor, Nov 2005.

 function xnew = Recip3(a);

     % Step 0 :  Special Cases should be at the start
         if a == 1
            xnew = a;
            return ;
         elseif a == 0
             xnew = inf;
             return ;
         elseif abs(a) == inf
             xnew = 0;
             return;
         end;
     % Step 1 :  Argument Reduction
         [f, e] = log2(a);
     % Step 2 :  Calculate Newton Approximation
         xold = 2^(-e);
         ek = 1 - a*xold;
         while ek > eps
             ek = 1 - a*xold;
             xnew = xold + ek*xold;
             xold = xnew;
         end;
     % Step 3  :  Reconstruction - not needed here
```

Further Analysis of the Reciprocal Algorithm

We did a general analysis of the reciprocal algorithm in Section 2.1 and found that it converges for $0 < x_0 < 2/a$, and that the error at iteration k was $e_k = ae_{k-1}^2$ (see equations (1.3) and (1.4) above). We now need to go further with this analysis, having chosen a starting value $x_0 = 2^{-e}$, where $a = f \times 2^e$ is an IEEE double precision floating point number.

It is possible to do a fairly detailed analysis of this algorithm because it is simple. It requires very little mathematics and just an elementary knowledge of binary floating point arithmetic. The third implementation above is straight-forward and uses no clever tricks. It is an attempt to show what goes on in a floating point processor. Most hardware implementations of the reciprocal operation use a combination of methods, including Newtons' method. Although a pure Newton algorithm is rarely implemented in practice, it is a very good starting point for many implementations.

We start with a count of the elementary operations performed inside the while–loop : 2 mults , 1 add , and 1 sub .

We assume calculations are performed in IEEE double precision with 53 bits precision, eps $= 2^{52}$, and that all arguments a are normalized, and in the range $2^{-1022} \le a < 2^{+1024}$. We ignore sub-normals for the moment.

The Recip3 function begins with $x_0 = 2^{-e}$, where $a = f \times 2^e$, and $1/2 \le f < 1$. Hence

$$e_0 = \frac{1/a - x_0}{1/a} = 1 - ax_0 = 1 - f \times 2^e \times 2^{-e} = 1 - f, \text{ where } 0 < x_0 \le 1/a.$$

The bounds on e_0 are, therefore,

$$\boxed{0 < e_0 \le 2^{-1}} \tag{1.7}$$

This means that x_0 has at least 1 bit of precision. Recip3, having 2nd-order convergence, will double the number of bits of precision at each iteration. Hence we can expect 6 iterations, $1 \to 2 \to 4 \to 8 \to 16 \to 32 \to 64$, to get full 53-bit precision.

Let us be more precise. We have from equation (3), $e_k = e_{k-1}^2$, and from (5) we have $e_k = (1 - ax_0)^{2^k}$, or $e_k = e_0^{2^k}$. The number of iterations needed to gain full precision is the k that satisfies

$$e_k = e_0^{2^k} \le \epsilon_m = 2^{-52}, \quad \text{or} \quad (2^{-1})^{2^k} = 2^{-2^k} \le 2^{-52}.$$

This inequality is satisfied when $2^k > 52$. Hence $k = \lceil \log_2 52 \rceil = 6$ iterations are needed for full precision. Looking back at the symbolic calculations in Section 2.2, we see that the first example confirms this analysis in a semi-numerical way.

Testing the Reciprocal Implementations

The first test is designed to show the effect of the starting value x_0 on the number of iterations to convergence. Figure 1.3 shows that a good starting value is important.

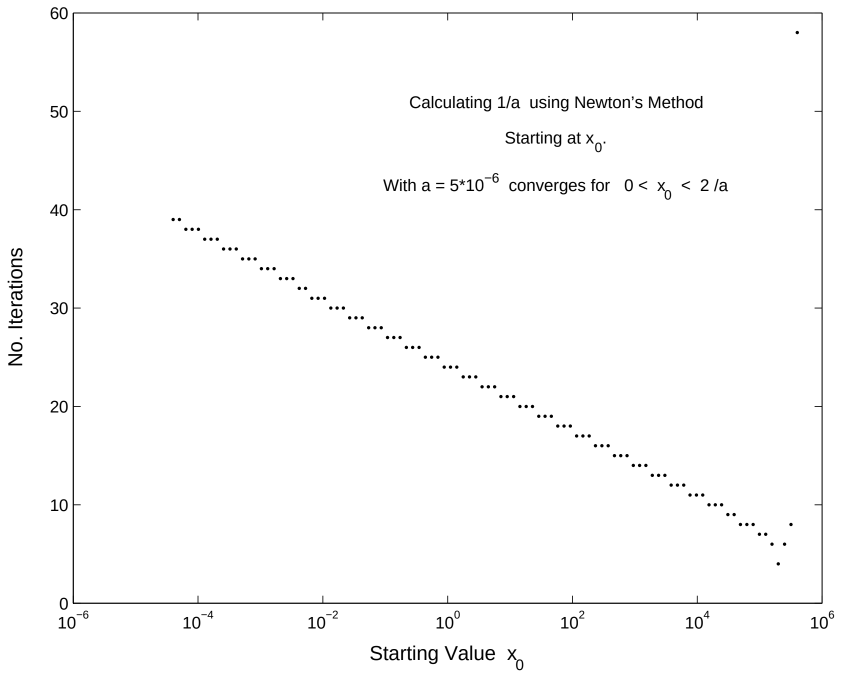

Figure 1.3 : Number of Iterations as a function of starting value x_0

The second test checks the accuracy of various implementations and the number of iterations to convergence, given a good starting value is used.

It can be seen in Figure 1.4 that, for a random sample of 1000 floating point numbers, Recip3(x) calculates $1/x$ to within ϵ_m of MATLAB'S value, over a broad range. This gives us some confidence in this function but random testing of this nature is rarely conclusive evidence of a function's correctness. There are about 10^{19} floating point numbers and so even very large random samples will miss most of these.

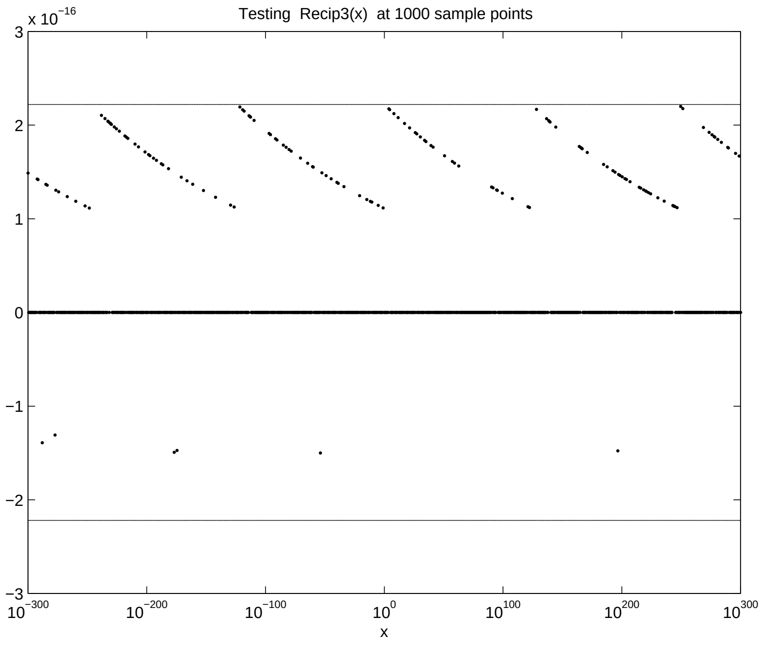

Figure 1.4 : Relative Error in Recip3. Average 6.25 iterations to converge.

Figure 1.5 shows the results for RecipM[5] which uses the following argument reduction and starting value :

```
r = a;
p = 0;
while r > 3/2
    p = p-1;
    r = r*0.5;
end;
while r < 3/4
    p = p+1;
    r = r*2;
end;            % 3/4 <= r <= 3/2

xold = sign(a)*2^(p)*(2-sign(a)*2^(p)*a);
```

[5]This is Peter Markstein's method. Ref ??

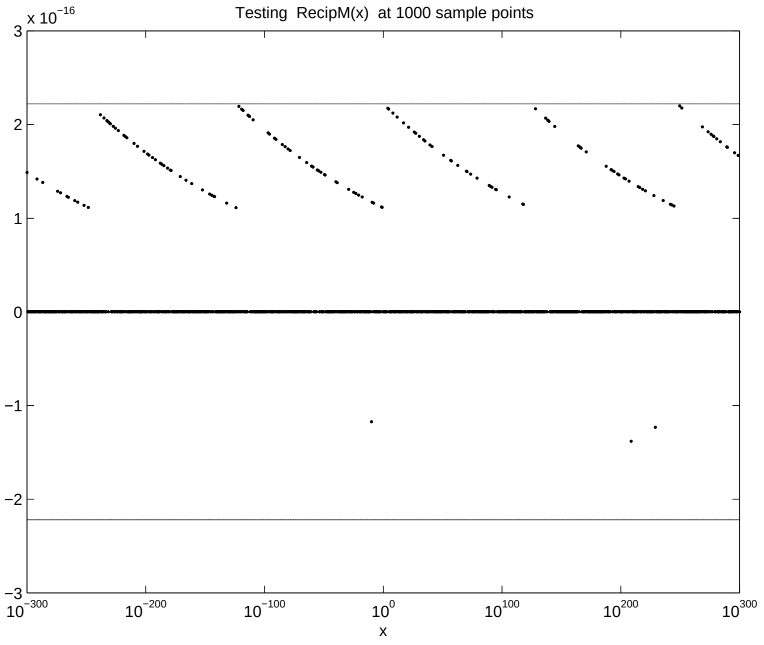

Figure 1.5 : Relative Error in RecipM. Average 4.84 iterations to converge.

1.3 NEWTON'S METHOD FOR SQUARE ROOT

We may define the square root of some number $a > 0$ as a zero of the function $f(x) = x^2 - a$. Using this function in Newton's method $x_{k+1} := x_k - f(x_k)/f'(x_k)$ we get[6]

$$\boxed{x_{k+1} := x_k - \frac{x_k^2 - a}{2x_k} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right) \triangleq T(x_k)} \tag{1.8}$$

Notice that if $x_k > \sqrt{a}$ then $a/x_k < \sqrt{a}$ and *vice versa.* Thus the next approximation x_{k+1} is half-way between these bounds.

The steps of Newton's method are shown graphically in 1.3

1.3.1 Analysis of Newton's Square Root Method

Obviously $\sqrt{a}$ is a fixed point of T because $T(\sqrt{a}) = (\sqrt{a} + a/\sqrt{a})/2 = \sqrt{a}$ Under what condition(s) will the iterative method $x_{k+1} := (x_k + a/x_k)/2 = T(x_k)$, converge to $\sqrt{a}$?

Again we use a method from the book by Demidovitch & Maron who express the error at the kth iteration in terms of the initial error.

[6]This is in fact a method used by *Heron of Alexandria* (10AD(?) – 75AD(?). See the note at the end of this chapter.

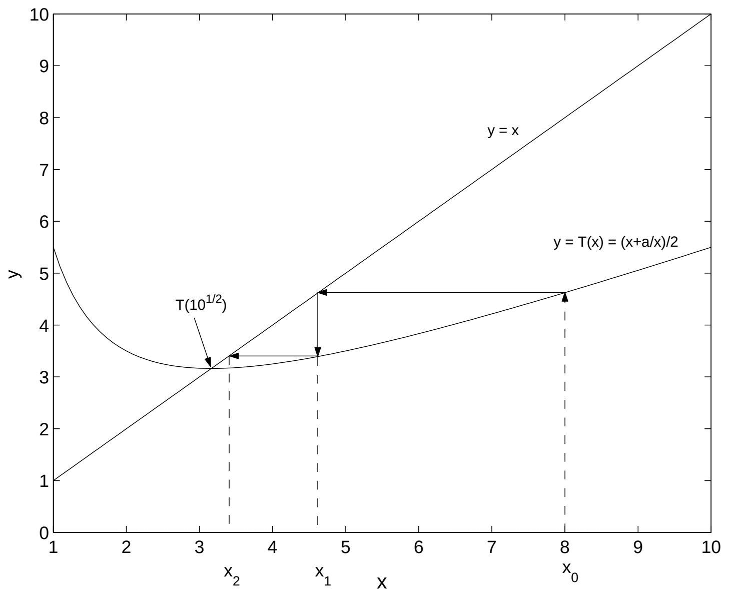

Figure 1.6 : Fixed Point Iterations of Newton's Method for $\sqrt{a}$

Let $e_k = |x_k - \sqrt{a}\,|$ be the error[7] at the kth iteration of $x_{k+1} := (x_k + a/x_k)/2$. Then we have

$$\begin{aligned} e_k = \left(x_k - \sqrt{a}\,\right) &= \left((x_{k-1} + a/x_{k-1})/2 - \sqrt{a}\,\right) \\ &= \frac{1}{2x_{k-1}}\left(x_{k-1}^2 - 2x_{k-1}\sqrt{a} + a\right) \\ &= \frac{1}{2x_{k-1}}\left(x_{k-1} - \sqrt{a}\,\right)^2 \\ &= \frac{1}{2x_{k-1}}\, e_{k-1}^2 \end{aligned}$$

This result is not quite what we want because of the factor $1/2x_{k-1}$. We get rid of it by using the trick in Demodovich & Maron, page 108. We have

$$\begin{aligned} \left(x_k - \sqrt{a}\,\right) &= \frac{1}{2x_{k-1}}\left(x_{k-1} - \sqrt{a}\,\right)^2. && \text{Substituting } \sqrt{a} \text{ for } -\sqrt{a} \text{ we get} \\ \left(x_k + \sqrt{a}\,\right) &= \frac{1}{2x_{k-1}}\left(x_{k-1} + \sqrt{a}\,\right)^2. && \text{Dividing across by this we get} \\ \frac{(x_k - \sqrt{a}\,)}{(x_k + \sqrt{a}\,)} &= \left(\frac{x_{k-1} - \sqrt{a}}{x_{k-1} + \sqrt{a}}\right)^2. && \end{aligned} \tag{1.9}$$

Using successive substitution we get

$$\frac{(x_k - \sqrt{a}\,)}{(x_k + \sqrt{a}\,)} = \left(\frac{x_0 - \sqrt{a}}{x_0 + \sqrt{a}}\right)^{2^k} \tag{1.10}$$

This sequence converges to 0 if, and only if, $(x_0 - \sqrt{a})/(x_0 + \sqrt{a}) < 1$. If $x_0 > 0$ then $(x_0 - \sqrt{a}) < (x_0 + \sqrt{a})$.

[7]We are not using the relative error here.

Therefore, the sequence converges for all $x_0 > 0$. Also

$$\lim_{k\to\infty} \frac{(x_k - \sqrt{a}\,)}{(x_k + \sqrt{a}\,)} = \frac{(x - \sqrt{a}\,)}{(x + \sqrt{a}\,)} = 0 \Rightarrow \lim_{k\to\infty} x_k = x = \sqrt{a}.$$

$$\boxed{\text{Newton's Algorithm converges to the fixed point } x = \sqrt{a}, \text{ for all } x_0 > 0.} \tag{1.11}$$

1.3.2 Using the Square Root Algorithm Symbolically

Let us use the square root algorithm symbolically. The iteration function is $x_{k+1} = (x_k + a/x_k)/2$, and we begin by defining it with the Maxima command :

%i62 T(x) := (x+a/x)/2;

%o62 $\mathrm{T}(x) := \frac{\frac{a}{x}+x}{2}$

%i63 [r:T(5*sqrt(a)),float(r)]; — Start with the value $x_0 = 5\sqrt{a} > 0$.

%o63 $\left[\frac{13}{5}\sqrt{a}, 2.6\sqrt{a}\right]$

%i64 [r:T(r), float(r)]; — Repeatedly apply T to r.

%o64 $\left[\frac{97}{65}\sqrt{a}, 1.492307692307692\sqrt{a}\right]$

%i65 [r:T(r), float(r)];

%o65 $\left[\frac{6817}{6305}\sqrt{a}, 1.081205392545599\sqrt{a}\right]$

%i66 [r:T(r), float(r)];

%o66 $\left[\frac{43112257}{42981185}\sqrt{a}, 1.00304952038898\sqrt{a}\right]$

%i67 [r:T(r), float(r)];

%o67 $\left[\frac{1853024483819137}{1853015893884545}\sqrt{a}, 1.00000463565079\sqrt{a}\right]$

%i68 [r:T(r), float(r)];

%o68 $\left[\frac{3433683820310959228731558640897}{3433683820274065740584139537665}\sqrt{a}, 1.000000000010745\sqrt{a}\right]$

%i69 [r:T(r), float(r)];

%o69 $\left[\frac{11790184577738583171521213143779439604141674966883272877308417}{11790184577738583171520532579045597727214748217668409340885505}\sqrt{a}, \sqrt{a}\right]$

This clearly shows the sequence converging to $\sqrt{a}$ in 7 iterations, with 16-digit precision. It is obvious from Figure 3.6 that after the first iteration the method approaches $\sqrt{a}$ from above. We prove this using the expression in (1.9) :

$$\frac{(x_k - \sqrt{a}\,)}{(x_k + \sqrt{a}\,)} = \left(\frac{x_{k-1} - \sqrt{a}}{x_{k-1} + \sqrt{a}}\right)^2 \Rightarrow \frac{(x_k - \sqrt{a}\,)}{(x_k + \sqrt{a}\,)} > 0.$$

Now, $x_k > 0$ and so $(x_k + \sqrt{a}) > 0$. This implies $(x_k - \sqrt{a}) > 0$ and so $x_k > \sqrt{a}$. We may have $x_0 < \sqrt{a}$ but for $k > 0$ we will have $x_k > \sqrt{a}$. We now have the bounds

$$\frac{a}{x_k} < \sqrt{a} < x_k, \tag{1.12}$$

and the new iterate x_{k+1} is half-way between these bounds.

1.3.3 Convergence Behavior

This section is based on notes by Schlomo Sternberg, Harvard University

Newton's method must be started sufficiently close to a zero of $f(x)$ if 2nd order convergence is to be achieved. If not, the method may not converge or it may converge slowly.

We have shown that Newton's method for $f(x) = x^2 - a$ converges to $\sqrt{a}$ all $x_0 > 0$. We now look at the convergence behavior of the method as a function of the starting point x_0. We need to get an expression for e_{k+1} in terms of e_k.

Let $e_k = (x_k - \sqrt{a})/\sqrt{a}$ be the relative error at iteration k. Thus $x_k = (1+e_k)\sqrt{a}$. Substitute this into the iteration function and we get

$$x_{k+1} = \frac{1}{2}\left((1+e_k)\sqrt{a} + \frac{a}{(1+e_k)\sqrt{a}}\right) = \frac{\sqrt{a}}{2}\left((1+e_k) + \frac{1}{(1+e_k)}\right)$$

Hence

$$e_{k+1} = x_{k+1} - \sqrt{a} = \frac{\sqrt{a}}{2}\left((1+e_k) + \frac{1}{(1+e_k)}\right) - \sqrt{a} = \frac{\sqrt{a}}{2}\left(e_k - 1 + \frac{1}{(1+e_k)}\right).$$

This simplifies to

$$\boxed{e_{k+1} = \frac{\sqrt{a}}{2}\left(\frac{e_k^2}{1+e_k}\right)} \tag{1.13}$$

We can now determine how Newton's square root algorithm behaves when far from or close to $\sqrt{a}$:

- If $x_k \gg \sqrt{a}$ then $e_k \gg 1$ and $e_{k+1} \approx \sqrt{a}e_k/2$ 1st-order convergence.
- If $x_k \approx \sqrt{a}$ then $e_k \ll 1$ and $e_{k+1} \approx \sqrt{a}e_k^2/2$ 2nd-order convergence.

This behavior is clearly seen in Table I, where convergence is 1st-order for the first 7 or 8 iterations, and becomes 2nd-order after iteration 11.

TABLE I : NEWTON SQUARE ROOT OF 2

k	x_{k-1}	e_k	e_k/e_{k-1}
1	1000	706.1067811865475244008443621	0.4992928932188134524755991557
2	500.00100000000000000000000000	352.5540977000549487479465819	0.4985857892660483728544526024
3	250.00249999600000799998400000	175.7784630607614260011188383	0.4971716011591969636418159377
4	125.0052499580004679945840631	87.39205992922151603691760324	0.4943433833265073044738395367
5	62.51062464301703314888691358	43.20168658128425071398496208	0.4886882144399506116918387102
6	31.27130960206219455596422359	21.11215507620217474530064233	0.4773880022875691401866075361
7	15.66763299486836640030755527	10.07868953581351823246371363	0.4548683083514818843615153083
8	7.897642347856358067190513609	4.584476459555277231870341506	0.4104660922789854919930748575
9	4.075441240519498920887985734	1.881772137498653123942095896	0.3264956505430042215285226191
10	2.283092824392553839863066903	0.614390418206322340442525329	0.1902855750621167308130014610
11	1.579548752406015365275470017	0.116909634041044439408261203	0.0523362098767375385616799203
12	1.422866579578668250912096839	0.006118607143784681142450681	0.0030406987309152654143791569
13	1.414239873591530623193646164	0.000018604840977075156846184	0.000009302247421703568221
14	1.414213562617848512655890004	0.000000000173066834010202273	0.0000000000865334170
15	1.414213562373095048822868078	0.000000000000000000014976065	0.0000000000

1.3.4 Implementation, Analysis, and Testing of the Square Root Algorithm

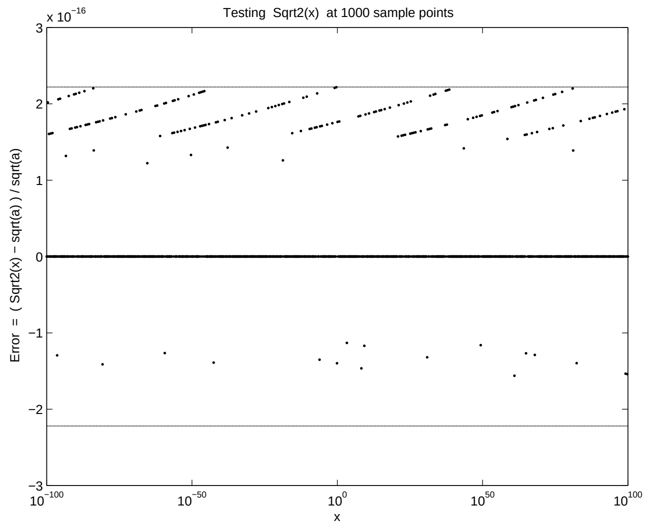

Figure 1.7 : Relative Error in Sqrt2. Average 7 iterations to converge

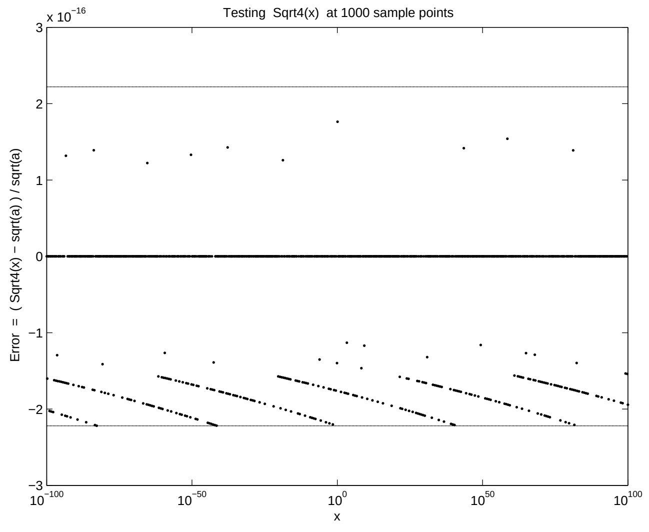

Figure 1.8 : Relative Error in Sqrt4. Average 4 iterations to converge

Further Analysis of the Square Root Algorithm

TO BE COMPLETED

1.4 THE RELATIONSHIP BETWEEN RECIPROCAL AND SQUARE ROOT

The algorithms for calculating the elementary operations or functions $F_1(a) = 1/a$, $F_2(a) = \sqrt{a}$, and $F_3(a) = 1/\sqrt{a}$, are very similar and are often used together. We will develop a Newton algorithm for each in *correction form*, i.e., $x_{k+1} := x_k + \Delta x_k$, in order to illuminate this similarity. Indeed, Newton's method, $x_{k+1} := x_k - f(x_k)/f'(x_k)$, is in correction form and we will try to preserve this in our development algorithms for the three operations above.

The choice of $f(x)$ used in Newton's method is arbitrary except that it must be chosen so that $f(x) = 0$, e.g., $f(1/x) = 0$. Here is a list of possible functions for the three

1. Reciprocal : $f_1(x) = 1/x - a$.
2. Square Root : $f_2(x) = x^2 - a$.
3. Reciprocal Square Root : $f_3(x) = 1/x^2 - a$.

We note, for future reference, that $F_3(a) = 1/\sqrt{a}$ can be used to compute other operations : $\sqrt{a} = 1/F_3(a)$, $1/a = F_3^2(a)$, and $a/b = aF_3^2(b)$. Note also that $a/b = aF_1(b)$.

1.4.1 Correction Form Algorithms

1. *The Reciprocal Newton* algorithm is

$$x_{k+1} := x_k - \frac{f_1(x)}{f_1'(x)} = x_k - \frac{1/x - a}{-1/x^2} = x_k + x_k(1 - ax_k).$$

We compute x_{k+1} in two steps

$$\boxed{\begin{array}{rcl} e_k & := & 1 - a \star x_k \\ x_{k+1} & := & x_k + e_k \star x_k \end{array}}$$

Notice that e_k is the relative error in x_k, i.e., $(1/a - x_k)/(1/a) = (1 - ax_k)$, and that $e_k x_k$ is the correction term. Calculating x_{k+1} requires 2 mults and 2 adds.

2. The *Square Root Newton* algorithm is

$$x_{k+1} := x_k - \frac{f_2(x)}{f_2'(x)} = x_k - \frac{x_k^2 - a}{2x_k} = x_k + (a - x_k^2)/2x_k.$$

We compute x_{k+1} in two steps

$$\boxed{\begin{array}{rcl} e_k & := & a - x_k \star x_k \\ x_{k+1} & := & x_k + e_k/(2 \star x_k) \end{array}}$$

The term $e_k = a - x_k^2$ is the absolute error in x_k^2 and $e_k/2x_k$ is the correction term. Calculating x_{k+1} requires 1 mult, 1 div, and 2 adds. Multiplication by 2 can be done by *bit-shifting*.

3. The *Reciprocal Square Root Newton* algorithm is

$$x_{k+1} := x_k - \frac{f_3(x)}{f_3'(x)} = x_k - \frac{1/x^2 - a}{-2/x^3} = x_k + \frac{x_k(1 - ax_k^2)}{2}.$$

We compute x_{k+1} in two steps

$$\boxed{\begin{array}{rcl} e_k & := & 1-a\star x_k\star x_k \\ x_{k+1} & := & x_k+e_k\star x_k/2 \end{array}}$$

Here, e_k is the relative error in x_k^2, i.e., $(1/a-x_k^2)/(1/a)=(1-ax_k^2)$, and $e_kx_k/2$ is the correction term. The most important aspect of this algorithm is that *no divisions* are required. The division by 2 can be done by bit-shifting. Calculating x_{k+1} requires 3 mults and 2 adds.

The reciprocal and reciprocal square root algorithms are remarkable similar

$$\boxed{\begin{array}{rclr} e_k & := & 1-a\star x_k & e_k := 1-a\star x_k\star x_k \\ x_{k+1} & := & x_k+e_k\star x_k & x_{k+1} := x_k+e_k\star x_k/2 \end{array}}$$

TO BE COMPLETED

1.5 HISTORICAL NOTES

1.5.1 Heron of Alexandria

This is part of a longer article by: J J O'Connor and E F Robertson, April 1999.
http://www-history.mcs.st-andrews.ac.uk/Mathematicians/Heron.html

Born: about 10AD in (possibly) Alexandria, Egypt Died: about 75AD

Sometimes called Hero, **Heron of Alexandria** was an important geometer and worker in mechanics. Perhaps the first comment worth making is how common the name Heron was around this time and it is a difficult problem in the history of mathematics to identify which references to Heron are to the mathematician described in this article and which are to others of the same name. There are additional problems of identification which we discuss below. A major difficulty regarding Heron was to establish the date at which he lived. There were two main schools of thought on this, one believing that he lived around 150 BC and the second believing that he lived around 250 AD. The first of these was based mainly on the fact that Heron does not quote from any work later than Archimedes. The second was based on an argument which purported to show that he lived later that Ptolemy, and, since Pappus refers to Heron, before Pappus.

Both of these arguments have been shown to be wrong. There was a third date proposed which was based on the belief that Heron was a contemporary of Columella. Columella was a Roman soldier and farmer who wrote extensively on agriculture and similar subjects, hoping to foster in people a love for farming and a liking for the simple life. Columella, in a text written in about 62 AD [5]:-

> ... gave measurements of plane figures which agree with the formulas used by Heron, notably those for the equilateral triangle, the regular hexagon (in this case not only the formula but the actual figures agree with Heron's) and the segment of a circle which is less than a semicircle ...

However, most historians believed that both Columella and Heron were using an earlier source and claimed that the similarity did not prove any dependence. We now know that those who believed that Heron lived around the time of Columella were in fact correct, for Neugebauer in 1938 discovered that Heron referred to a recent eclipse in one of his works which, from the information given by Heron, he was able to identify with one which took place in Alexandria at 23.00 hours on 13 March 62. From Heron's writings it is reasonable to deduce that he taught at the Museum in Alexandria. His works look like lecture notes from courses he must have given there on mathematics, physics, pneumatics, and mechanics. Some are clearly textbooks while others are perhaps drafts of lecture notes not yet worked into final form for a student textbook. Pappus describes the contribution of Heron in Book VIII of his *Mathematical Collection*. Pappus writes (see for example [8]):-

The mechanicians of Heron's school say that mechanics can be divided into a theoretical and a manual part; the theoretical part is composed of geometry, arithmetic, astronomy and physics, the manual of work in metals, architecture, carpentering and painting and anything involving skill with the hands.

... the ancients also describe as mechanicians the wonder-workers, of whom some work by means of pneumatics, as Heron in his Pneumatica, some by using strings and ropes, thinking to imitate the movements of living things, as Heron in his Automata and Balancings, ... or by using water to tell the time, as Heron in his Hydria, which appears to have affinities with the science of sundials.

A large number of works by Heron have survived, although the authorship of some is disputed. We will discuss some of the disagreements in our list of Heron's works below. The works fall into several categories, technical works, mechanical works and mathematical works. The surviving works are: *On the dioptra* dealing with theodolites and surveying. It contains a chapter on astronomy giving a method to find the distance between Alexandria and Rome using the difference between local times at which an eclipse of the moon is observed at each cities. The fact that Ptolemy does not appear to have known of this method led historians to mistakenly believe Heron lived after Ptolemy; *The pneumatica* in two books studying mechanical devices worked by air, steam or water pressure. It is described in more detail below; *The automaton theatre* describing a puppet theatre worked by strings, drums and weights; *Belopoeica* describing how to construct engines of war. It has some similarities with work by Philon and also work by Vitruvius who was a Roman architect and engineer who lived in the 1st century BC; *The cheirobalistra* about catapults is thought to be part of a dictionary of catapults but was almost certainly not written by Heron; *Mechanica* in three books written for architects and described in more detail below; *Metrica* which gives methods of measurement. We give more details below; *Definitiones* contains 133 definitions of geometrical terms beginning with points, lines etc. In [15] Knorr argues convincingly that this work is in fact due to Diophantus; *Geometria* seems to be a different version of the first chapter of the *Metrica* based entirely on examples. Although based on Heron's work it is not thought to be written by him; *Stereometrica* measures three-dimensional objects and is at least in part based on the second chapter of the Me*trica* again based on examples. Again it is though to be based on Heron's work but greatly changed by many later editors; *Mensurae* measures a whole variety of different objects and is connected with parts of *Stereometrica* and *Metrica* although it must be mainly the work of a later author; *Catoprica* deals with mirrors and is attributed by some historians to Ptolemy although most now seem to believe that this is a genuine work of Heron. In this work, Heron states that vision results from light rays emitted by the eyes. He believes that these rays travel with infinite velocity. Let us examine some of Heron's work in a little more depth. Book I of his treatise *Metrica* deals with areas of triangles, quadrilaterals, regular polygons of between 3 and 12 sides, surfaces of cones, cylinders, prisms, pyramids, spheres etc. A method, known to the Babylonians 2000 years before, is also given for approximating the square root of a number. Heron gives this in the following form (see for example [5]):-

> Since 720 has not its side rational, we can obtain its side within a very small difference as follows. Since the next succeeding square number is 729, which has 27 for its side, divide 720 by 27. This gives $26\frac{2}{3}$. Add 27 to this, making $53\frac{2}{3}$, and take half this or $26\frac{5}{6}$. The side of 720 will therefore be very nearly $26\frac{5}{6}$. In fact, if we multiply $26\frac{5}{6}$ by itself, the product is $720\frac{1}{36}$, so the difference in the square is $\frac{1}{36}$. If we desire to make the difference smaller still than $\frac{1}{36}$, we shall take $720\frac{1}{36}$ instead of 729 (or rather we should take $26\frac{5}{6}$ instead of 27), and by proceeding in the same way we shall find the resulting difference much less than $\frac{1}{36}$.

Heron also proves his famous formula in Book I of the *Metrica*:

> if A is the area of a triangle with sides a, b and c and $s = (a+b+c)/2$ then
>
> $$A^2 = s(s-a)(s-b)(s-c)$$

In Book II of *Metrica*, Heron considers the measurement of volumes of various three dimensional figures such as spheres, cylinders, cones, prisms, pyramids etc. His preface is interesting, partly because knowledge of the work of Archimedes does not seem to be as widely known as one might expect (see for example [5]):-

> After the measurement of surfaces, rectilinear or not, it is proper to proceed to solid bodies, the surfaces of which we have already measured in the preceding book, surfaces plane and spherical, conical and cylindrical, and irregular surfaces as well. The methods of dealing with these solids are, in view of their surprising character, referred to Archimedes by certain writers who give the traditional account of their origin. But whether they belong to Archimedes or another, it is necessary to give a sketch of these results as well.