Newton Iteration

Contents

1 Introduction

Newton iteration is the basic method to determine zeros of univariate and multivariate functions. I only discuss the univariate case. Let f be a real-valued function of a single variable; f is assumed to be continuously differentiable. The goal is to find a zero x^* of f. Consider an arbitrary value x_0. The tangent to f at $(x_0, f(x_0))$ has the equation

$$y = f(x_0) + (x - x_0)f'(x_0)$$

and intersects the x-axis in

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \; .$$

If x_0 is close to x^* and f is reasonably smooth in the vicinity of x^*, x_1 will be even closer to x^*. This suggests the iteration

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad \text{for } k \geq 0 \; .$$

The iteration above is called *Newton Iteration.*

In Section 2 show that Newton iteration converges quadratically to x^* if the initial approximation x_0 is sufficiently close to x^*. It is important to remember that Newton's method is not globally convergent. It needs a good initial approximation.

In Section 3 we will study the important case of computing square roots.

In Sections 4 and 5 we will leave the ideal world of exact arithmetic and take rounding error into account.

Section 6 remains to be written.

2 A Convergence Theorem

Theorem 1 (Deuflhardt/Hohman, Theorem 4.10, adapted to univariate case) *Let $I = [a\,..\,b]$ and let $f : I \mapsto \mathbb{R}$ be a continuously differentiable function with non-vanishing derivate for all $x \in \mathbb{R}$. Assume further that there is a constant*[1] [2] *ω such that for all $x, y \in I$*

$$\left|\frac{f'(y) - f'(x)}{f'(x)}\right| \leq \omega |y - x|$$

Assume further that there is an $x^* \in I$ with $f(x^*) = 0$ and that our initial approximation x_0 satisfies:

$$x_0 \in B_\rho(x^*) \subset I \quad \textit{where } \rho < 1/\omega \ .$$

Then $\{x_k\}$ converges quadratically to x^*, all iterates are contained in B_ρ, and x^* is the only root of f in B_ρ. More precisely,

$$x_k \in B_\rho \quad \textit{for all } k \quad \textit{and} \quad |x_{k+1} - x^*| \leq \frac{\omega}{2}|x_k - x^*|^2 \quad \textit{and} \quad |x_k - x^*| \leq \frac{\delta^{(2^k)}}{2^k\omega}$$

where $\delta = \rho/\omega < 1$.

Proof: Assume that $x_k \in B_\rho$. Then

$$\begin{aligned}
|x_{k+1} - x^*| &= |x_k - \frac{f(x_k)}{f'(x_k)} - x^*| \\
&= |x_k - x^* - \frac{f(x_k) - f(x^*)}{f'(x_k)}| \\
&= |x_k - x^* + \frac{f'(\xi)(x_k - x^*)}{f'(x_k)}|
\end{aligned}$$

where ξ is a point between x_k and x^*; recall that $(f(x_k) - f(x^*))/(x_k - x^*) = f(\xi)$ for some ξ between x_k and x^* by the mean value theorem.

$$\begin{aligned}
&= \left|\frac{f'(x_k) - f'(\xi)}{f'(x_k)}\right| |x_k - x^*| \\
&\leq \omega |(\xi - x^*)(x_k - x^*)| \leq \omega |x_k - x^*|^2 \ .
\end{aligned}$$

[1] Newton's method applied to $af(x)$ instead of $f(x)$ for an arbitrary constant a produces the same sequence of iterates. The requirement on ω is invariant under multiplication by constants.

[2] If f is twice differentiable, we can use $\omega = \frac{\max_{x\in I} f''(x)}{\min_{x\in I} f'(x)}$ since $f'(y) - f'(x) = f''(\xi)$ for some ξ between x and y by the mean value theorem.

This falls a factor 1/2 short of what we claimed. A slightly more elaborate argument gives us the improved bound. For any s with $0 \le s \le 1$, let $g(s) = f(x^* + s(x_k - x^*))$. Then

$$\begin{aligned}
f(x_k) - f(x^*) = g(1) - g(0) &= \int_0^1 g'(s)\ ds \\
&= \int_0^1 f'(x^* + s(x_k - x^*))(x_k - x^*)\ ds
\end{aligned}$$

and hence

$$\begin{aligned}
|x_{k+1} - x^*| &= |x_k - \frac{f(x_k)}{f'(x_k)} - x^*| \\
&= |x_k - x^* - \frac{f(x_k) - f(x^*)}{f'(x_k)}| \\
&= |\frac{f'(x_k)(x_k - x^*) + \int_0^1 f'(x^* + s(x_k - x^*))(x_k - x^*)\ ds}{f'(x_k)}| \\
&= |(x_k - x^*) \cdot \int_0^1 \frac{f'(x_k) + f'(x^* + s(x_k - x^*))}{f'(x_k)}\ ds| \\
&\le |(x_k - x^*) \cdot \int_0^1 \omega s (x_k - x^*)\ ds| \\
&\le \omega \int_0^1 s\ ds |x_k - x^*|^2 = \frac{\omega}{2} |x_k - x^*|^2
\end{aligned}$$

The remaining claims follow easily. We have

$$|x_0 - x^*| = \rho = \delta/\omega = \frac{\delta^{(2^0)}}{2^0 \omega}\ .$$

Also

$$\begin{aligned}
|x_{k+1} - x^*| &= \frac{\omega}{2} |x_k - x^*|^2 \\
&\le \frac{\omega}{2} \left(\frac{\delta^{(2^k)}}{2^k \omega} \right)^2 \\
&= \frac{(\delta^{(2^k)})^2}{2^{k+1} \omega} = \frac{\delta^{(2^k) \cdot 2}}{2^{k+1} \omega} = \frac{\delta^{(2^{k+1})}}{2^{k+1} \omega}
\end{aligned}$$

Uniqueness of x^* also follows easily. Assume the existence of a second root x^{**} in $B_\rho(x^*)$. Both points are fixed points of the Newton iteration. However, the Newton iteration started at one of the fix points must converge to the other. Thus there is only one fix point. ∎

We also need an estimate on the decrease of $|x_k - x_{k-1}|$, an observable quantity.

Lemma 1 *Under the assumptions of Theorem 1*

$$|x_{k+1} - x_k| \le 6\omega |x_k - x_{k-1}|^2\ .$$

Proof: We have $|x_{k+1} - x^*| \leq \omega/2|x_k - x^*| \leq 1/2|x_k - x^*|$ since $x_k \in B_\rho(x^*)$ and $\rho < 1/\omega$. Thus $|x_{k+1} - x_k| \leq |x_{k+1} - x^*| + |x^* - x_k| \leq (3/2)|x^* - x_k|$ and $|x_{k+1} - x_k| \geq |x_k - x^*| - |x^* - x_{k+1}| \geq (1/2)|x^* - x_k|$. Hence

$$|x_{k+1} - x_k| \leq \frac{3}{2}|x^* - x_k| \leq \frac{6\omega}{4}|x^* - x_{k-1}|^2 \leq \frac{6\omega}{4}\left(2|x_k - x_{k-1}|\right)^2 = 6\omega|x_k - x_{k-1}|^2 \ .$$

Observe that we use the contraction property for two subsequent iterations: the iteration from x_{k-1} to x_k and the iteration from x_k to x_{k+1}. ∎

3 The Computation of Square Roots

In this section, we will study the use of Newton Iteration to compute square roots. The goal is to compute $\sqrt{a}$ for a positive real a. We write $a = a'2^{2m}$ where $1/4 < a' \leq 1$ and $m \in \mathbb{Z}$. Then $\sqrt{a'}2^m$. So from now on we assume $1/4 < a \leq 1$.

The square root of a is a zero of $f(x) = x^2 - a$. We have $f'(x) = 2x$ and hence use the iteration formula

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

We use $I = [1/4 \,..\, 1]$. For $x, y \in I$, we have $|(f'(y) - f'(x))/f'(x)| = |(y-x)/x| \leq 4(y-x)$. Thus we may use $\omega = 4$ and hence need to start with an approximation x_0 with $x_0 - \sqrt{a} < 1/4$. We can use $x_0 = 1$ if $\sqrt{a} > 3/4$ or $a > 9/16$ and $x_0 = 0.6$ if $1/2 \leq \sqrt{a} \leq 3/4$.

We can obtain slightly stronger results if we study the iteration directly without relying on Theorem 1.

Theorem 2 *Let $1/4 \leq a \leq 1$ and let $x_0 = 1$. Then $\{x_k\}$ is monotonically decreasing, $\sqrt{a} \leq x_k \leq 1$ for all k, $x_{k+1} - \sqrt{a} \leq (x_k - \sqrt{a})^2$, and $x_k - \sqrt{a} \leq 2^{-(2^k)}$.*

Proof: Assume $\sqrt{a} \leq x_k \leq 1$. set $d_k = x_k - \sqrt{a}$. We have

$$\begin{aligned} d_{k+1} = x_{k+1} - \sqrt{a} &= \frac{1}{2}(x_k + \frac{a}{x_k}) - \sqrt{a} \\ &= \frac{1}{2x_k}(x_k^2 + a - 2\sqrt{a}x_k) \\ &= \frac{1}{2x_k}(x_k - \sqrt{a})^2 = \frac{1}{2x_k}d_k^2 \ . \end{aligned}$$

Thus $d_k^2/2 \leq d_{k+1} \leq d_k^2$ since $1/2 \leq 1/(2x_k) \leq 1$. In particular, $d_{k+1} \geq 0$ and $d_{k+1} \leq d_k/2$ and hence $\{x_k\}$ is converging from above to $\sqrt{a}$.

Also $d_0 \leq 1/2 = 2^{-(2^0)}$ and hence

$$d_{k+1} \leq d_k^2 \leq \left(2^{-(2^k)}\right)^2 = 2^{-(2^{k+1})} \ .$$

$\blacksquare$

We can also improve our statement about the observable quantities.

Lemma 2 *Assume* $\sqrt{a} \le x_k \le 1$. *Then*

$$|x_{k+1} - x_k| \le 4|x_k - x_{k-1}|^2 \ .$$

Proof: As in the preceding Lemma let $d_k = x_k - \sqrt{a}$. Then $d_{k+1} \le d_k^2$ by the preceding Lemma and hence

$$x_k - x_{k+1} \le x_k - \sqrt{a} = d_k \le d_{k-1}^2 \ .$$

It therefore suffices to show $d_{k-1} \le 2(d_{k-1} - d_k)$ or $d_k \le d_{k-1}/2$. This follows from $d_k \le d_{k-1}^2 \le d_{k-1}/2$. $\blacksquare$

4 Square Roots with Floating Point Arithmetic

We assume that we use floating point arithmetic with relative precision ϵ, i.e., the computed result of any operation differs from the true result by at most ϵ times the exact result. In other words, for any $\circ \in \{+, -, *, /\}$ we have

$$|x \tilde{\circ} y - x \circ y| \le \epsilon \cdot (x \circ y) \ .$$

or

$$x \tilde{\circ} y = (x \circ y)(1 + \delta) \quad \text{for some } \delta \text{ with } |\delta| \le \epsilon \ .$$

In the following we use $\oplus$, $\ominus$, $\otimes$, $\oslash$ to denote floating point operations.

Newton Iteration with floating point arithmetic produces a sequence $(\widehat{x}_k)_{k \in \mathbb{N}}$ of iterates, where $\widehat{x}_0 = 1$ and

$$\widehat{x}_{k+1} = (\widehat{x}_k \oplus (a \oslash \widehat{x}_k)) \oslash 2 = (\widehat{x}_k \oplus (a \oslash \widehat{x}_k))/2 \ .$$

Observe that a division by two incurs no round-off error. It amounts to a simple decrement of the exponent. We use x_{k+1} to denote *the exact value of the next iterate*, i.e.,

$$x_{k+1} = (\widehat{x}_k + (a/\widehat{x}_k))/2 = \sqrt{a} + \frac{1}{2\widehat{x}_k}(\widehat{x}_k - \sqrt{a})^2 \ . \tag{1}$$

We also define $x_0 = 1$. Note that the sequence $(x_k)_{k \in \mathbb{N}}$ is not the sequence obtained by an exact Newton iteration, i.e., our notation differs from the one used in the preceding section.

We need to discuss the following issues:

- What is the distance between $\widehat{x}_{k+1}$ and x_{k+1}?
- What is an appropriate termination condition?
- Will we always terminate?
- How good an approximation to $\sqrt{a}$ is the last iterate?
- Is there an a posteriori estimate for the quality of an approximation.
- How does one choose the working precision as a function of the desired quality of the result.

We discuss the items in term.

We first show[3]

$$|\widehat{x}_k - x_k| \le (10/4)\epsilon \quad \text{and} \quad \sqrt{a}/2 \le \widehat{x}_k \le 4/3$$

for all k. Observe that the iterates of the exact Newton iteration lie between $\sqrt{a}$ and 1. For the iterates of the approximate Newton iteration, we have only weaker inclusions. The claims are trivially true for $k = 0$. So assume the claim holds for some k. From equation (1) we conclude $\sqrt{a} \le x_{k+1} \le \sqrt{a} + (1/\sqrt{a})(\sqrt{a}/2)^2 = (5/4)\sqrt{a} \le 5/4$. We next bound $|\widehat{x}_{k+1} - x_{k+1}|$. We have

$$\begin{aligned}
\widehat{x}_{k+1} &= (\widehat{x}_k \oplus (a \oslash \widehat{x}_k)) \oslash 2 \\
&= (\widehat{x}_k + (a/\widehat{x}_k)(1+\delta_1))(1+\delta_2)/2
\end{aligned}$$

with $|\delta_i| \le \epsilon$ for $i = 1, 2$. Observe that a division by two incurs no round-off error since it amounts to a decrement of the exponent.

$$\begin{aligned}
&= (\widehat{x}_k + (a/\widehat{x}_k) + (a/\widehat{x}_k)\delta_1))(1+\delta_2)/2 \\
&= x_{k+1} + x_{k+1}\delta_2 + (a/\widehat{x}_k)\delta_1(1+\delta_2)/2\ .
\end{aligned}$$

Thus

$$\begin{aligned}
|\widehat{x}_{k+1} - x_{k+1}| &\le x_{k+1}\epsilon + (a/\widehat{x}_k)\epsilon(1+\epsilon_2)/2 = (x_{k+1} + (a/\widehat{x}_k)(1+\epsilon_2)/2)\cdot\epsilon \\
&\le ((5/4) + 2\sqrt{a}(1+\epsilon)/2)\epsilon \le (9/4+\epsilon)\epsilon \le (10/4)\epsilon
\end{aligned}$$

where the last inequality assumes $\epsilon \le 1/4$ and hence

$$\widehat{x}_{k+1} \le x_{k+1} + (10/4)\epsilon \le 5/4 + (10/4)\epsilon \le 4/3$$

[3] Our estimates are crude and can be improved. However, it took me several hours to write this section and I got tired at the end.

where the last inequality assume $\epsilon \le 1/32$ and

$$\widehat{x}_{k+1} \ge x_{k+1} - (10/4)\epsilon \ge \sqrt{a} - (10/4)\epsilon \ge \sqrt{a}/2 \ .$$

With exact arithmetic, Newton's method converges quadratically and the distance between subsequent iterates goes to zero quadratically. We now have an error of at most $(10/4)\epsilon$ in the computation of the iterates. We should therefore expect that the convergence breaks down once the distance between adjacent iterates comes close to $(10/4)\epsilon$. We therefore terminate once

$$|\widehat{x}_{k+1} \ominus \widehat{x}_k| \le \beta\epsilon$$

for a constant β still to be determined. For what choice of β is termination guaranteed? We have

$$\begin{aligned}
|\widehat{x}_{k+1} - \sqrt{a}| &\le (10/4)\epsilon + |x_{k+1} - \sqrt{a}| \\
&\le (10/4)\epsilon + |\widehat{x}_k - \sqrt{a}|^2 \\
&\le (10/4)\epsilon + (5/6)|\widehat{x}_k - \sqrt{a}|
\end{aligned}$$

since $|\widehat{x}_k - \sqrt{a}| \le 5/6$ (Observe: if $\widehat{x}_k \le 1/2$ and hence $\widehat{x}_k \le \sqrt{a}$, we have $|\widehat{x}_k - \sqrt{a}| \le \sqrt{a}/2 \le 1/2$. If $\widehat{x}_k \ge 1/2$, we have $\widehat{x}_k, \sqrt{a} \in [1/2, 4/3]$ and hence $|\widehat{x}_k - \sqrt{a}| \le 5/6$.)

$$\le (11/12)|\widehat{x}_k - \sqrt{a}|$$

provided that $(1/12)|\widehat{x}_k - \sqrt{a}| \ge (10/4)\epsilon$ or $|\widehat{x}_k - \sqrt{a}| \ge 30\epsilon$. The iteration is therefore guaranteed to reach a $\widehat{x}_k$ with $|\widehat{x}_k - \sqrt{a}| < 30\epsilon$. The argument above also, shows that once $|\widehat{x}_k - \sqrt{a}| < 30\epsilon$, this will stay true forever. Then

$$\begin{aligned}
|\widehat{x}_{k+1} \ominus \widehat{x}_k| &\le |\widehat{x}_{k+1} - \widehat{x}_k|(1+\epsilon) \\
&\le (|\widehat{x}_{k+1} - x_{k+1}| + |x_{k+1} - \sqrt{a}| + |\sqrt{a} - \widehat{x}_k|) \cdot (1+\epsilon) \\
&\le [(10/4)\epsilon + |\widehat{x}_k - \sqrt{a}|^2 + |\widehat{x}_k - \sqrt{a}|](1+\epsilon) \\
&\le [(10/4)\epsilon + (30\epsilon)^2 + 30\epsilon](1+\epsilon) \\
&< 33\epsilon \ .
\end{aligned}$$

Thus $\beta = 33\epsilon$ guarantees termination.

We finally come to the question, we are really interested in. How good an approximation of $\sqrt{a}$ do we have at termination? We have $|\widehat{x}_{k+1} - \widehat{x}_k| \le 33\epsilon$ and hence $|x_{k+1} - \widehat{x}_k| \le |x_{k+1} - \widehat{x}_{k+1}| + |\widehat{x}_{k+1} - \widehat{x}_k| \le 36\epsilon$. Thus

$$\begin{aligned}
|\sqrt{a} - \widehat{x}_k| &\le |\sqrt{a} - x_{k+1}| + |x_{k+1} - \widehat{x}_k| \\
&\le |\sqrt{a} - \widehat{x}_k|^2 + 36\epsilon
\end{aligned}$$

Let $\delta = |\sqrt{a} - \widehat{x}_k|$. Then $\delta \le \delta^2 + 36\epsilon$ or $(\delta - 1/2)^2 \ge 1/4 - 36\epsilon = 1/4(1 - 144\epsilon$. Thus $\delta \le 1/2 - 1/2\sqrt{1 - 144\epsilon} \le 1/2 - 1/2(1 - 73\epsilon) \le 37\epsilon$. For the last inequality observe, that $(1 - 37\epsilon)^2 = 1 - 74\epsilon + 37^2\epsilon^2 \le 1 - 73\epsilon$ provided that $\epsilon \le 1/37$.

We summarize the discussion

- We use floating point arithmetic with precision ϵ with $\epsilon \le 1/37$. This is not a restriction; already double precision floating point arithmetic gives us $\epsilon = 2^{-53}$.
- We set $x_0 = 0$ and $x_{k+1} = (x_k \oplus a \oslash x_k)/2$.
- We terminate as soon as $|x_{k+1} \ominus x_k| \le 33\epsilon$.
- At termination we have $|x_k - \sqrt{a}| \le 37\epsilon$.
- If we want to guarantee maximal error 2^{-m} we choose $\epsilon = 2^{-(m+6)}$. Observe that $2^6 = 64 \ge 37$.

We next estimate the running time. Assume that we want to guarantee an error of 2^{-m}. Then we work with mantissa length $m + 6$ and hence the cost of an iteration is $O(M(m))$, where $M(m)$ is the time to divide m bit integers. LEDA uses Karazuba-Offman multiplication with $M(m) = m^{\log 3}$. Newton's method converges quadratically and hence the total running time is $O(M(m) \log m)$.

Many of our estimates are fairly crude and we might actually compute an estimate that is better than what we claim (not much better; we should not expect an error that is less than ϵ and hence our estimates are off by at most the factor 37). We next show that it is easy to give an a posteriori estimate of the quality of the final iterate. We have

$$|x - \sqrt{a}| = \frac{|(x - \sqrt{a})(x + \sqrt{a})|}{|x + \sqrt{a}|} \le 2|x^2 - a| \ .$$

We close this section we two test programs which illustrate Newton

' s method.

Quality of Final Iterate: A Demo Program: We implement the method as described above and output the claimed quality and the actual quality of the final iterate.

⟨*Newton_test.c*⟩≡

```
#include <LEDA/bigfloat.h>
main(){
while (true)
{ int m = read_int("\n\n\nmaximal error less than 2^{-m}; m = ");
double a = read_real("\na double value between 1/4 and 1, a =  ");
```

```
float T = used_time();

bigfloat::set_precision(m + 6);
bigfloat::set_output_precision(m + 6);

bigfloat xold;
bigfloat xnew = 1;
bigfloat A(a);
bigfloat epsilon = bigfloat(1,-m -6);
bigfloat Epsilon = 33*epsilon;
int iterations = 0;

do{ xold = xnew;
    xnew = (xold + A/xold)/2;
    iterations++;
} while ( abs( xold - xnew ) > Epsilon );

cout << "\n\nclaimed error is less than 2^L where L = " << -m ;

bigfloat::set_precision(4*m);

bigfloat est = 2.0 * abs(xnew*xnew - A);

cout << "\nactual error is less than 2^L where L = " << ilog2(est);

cout <<"\nnumber of iterations = " << iterations;

cout << "\nthis took " << used_time(T) << " seconds";

}

return 0;
}
```

The following is a sample run of the program. We see that the actual error is slightly better than what we claim.

```
maximal error less than 2^{-m}; m = 100000

a double value between 1/4 and 1, a =  0.56543254

claimed error is less than 2^L where L = -100000
actual error is less than 2^L where L = -100004
number of iterations = 17
this took 1.79 seconds
```

Quadratic Convergence: We implemented the method described above. In each iteration we report the quality of the iterate. This nicely shows the quadratic convergence of the method.

$\langle$*Newton_demo.c*$\rangle\equiv$

```
#include <LEDA/bigfloat.h>

main(){

while (true)
```

```
{ int m = read_int("\n\n\nmaximal error less than 2^{-m}; m = ");
double a = read_real("\na double value between 1/4 and 1, a =  ");
float T = used_time();
bigfloat::set_precision(m + 6);
bigfloat::set_output_precision(m + 6);
bigfloat xold;
bigfloat xnew = 1;
bigfloat A(a);
bigfloat epsilon = bigfloat(1,-m -6);
bigfloat Epsilon = 33*epsilon;
int iterations = 0;
do{ bigfloat::set_rounding_mode(TO_INF);
    bigfloat::set_precision(4*m);
    bigfloat est = 2.0 * abs(xnew*xnew - A);
    cout << "\n\niteration = " << iterations;
    if ( est == 0 ) cout << "   actual error is zero";
    else cout << "   actual error is less than 2^L where L = " << ilog2(est);
    bigfloat::set_rounding_mode(TO_NEAREST);
    bigfloat::set_precision(m + 6);
    xold = xnew;
    xnew = (xold + A/xold)/2;
    iterations++;
} while ( abs( xold - xnew ) > Epsilon );
cout << "\n\nthis took " << used_time(T) << " seconds";
}
return 0;
}
```

The following is a sample run of the program.

```
maximal error less than 2^{-m}; m = 100000
a double value between 1/4 and 1, a =  0.56543254
iteration = 0   actual error is less than 2^L where L = 0
iteration = 1   actual error is less than 2^L where L = -3
iteration = 2   actual error is less than 2^L where L = -9
iteration = 3   actual error is less than 2^L where L = -20
iteration = 4   actual error is less than 2^L where L = -42
iteration = 5   actual error is less than 2^L where L = -88
iteration = 6   actual error is less than 2^L where L = -178
iteration = 7   actual error is less than 2^L where L = -358
iteration = 8   actual error is less than 2^L where L = -719
```

```
iteration = 9   actual error is less than 2^L where L = -1441
iteration = 10   actual error is less than 2^L where L = -2885
iteration = 11   actual error is less than 2^L where L = -5773
iteration = 12   actual error is less than 2^L where L = -11549
iteration = 13   actual error is less than 2^L where L = -23101
iteration = 14   actual error is less than 2^L where L = -46205
iteration = 15   actual error is less than 2^L where L = -92412
iteration = 16   actual error is less than 2^L where L = -100004
this took 2.02 seconds
```

5 Adaptive Precision

Our implementation of Newton's method is very wasteful. In this section we will learn how to improve it by the method of *adaptive precision.*

The idea is simple. We start with a working precision of, say, 2^{-16}, and start Newton's method. When our iterates stop to improve, we double the exponent of the working precision (if this brings us above $m+6$ we take $m+6$) and continue. In this way, only the last iterations use the full precision and hence we gain enormously.

We have only a constant number of iterations for each working precision and hence total running time is $O(\sum_{1\le i\le\lceil\log m\rceil} M(2^i)) = O(M(m))$. We thus safe a $\log m$-factor.

$\langle$*Improved_Newton_demo.c*$\rangle\equiv$

```
#include <LEDA/bigfloat.h>
main(){
while (true)
{ int m = read_int("\n\n\nmaximal error less than 2^{-m}; m = ");
double a = read_real("\na double value between 1/4 and 1, a =  ");
float T = used_time();
int ma = 16;  // actual precision
bigfloat::set_precision(ma);
bigfloat xold;
bigfloat xnew = 1;
bigfloat A(a);
int iterations = 0;
while ( true )
{ bigfloat epsilon = bigfloat(1,-ma - 6);
  bigfloat Epsilon = 33*epsilon;
  do{
      xold = xnew;
      xnew = (xold + A/xold)/2;
```

```
        iterations++;
        bigfloat::set_rounding_mode(TO_INF);
        bigfloat::set_precision(4*ma);
        bigfloat est = 2.0 * abs(xnew*xnew - A);
        cout << "\n\niteration = " << iterations;
        if ( est == 0 ) cout << "   actual error is zero";
        else cout << "   actual error is less than 2^L where L = " << ilog2(est);
        bigfloat::set_rounding_mode(TO_NEAREST);
        bigfloat::set_precision(ma);
    } while ( abs( xold - xnew ) > Epsilon );
    if ( ma >= m + 8 ) break;
    ma = leda_min(2*ma,m+8);  bigfloat::set_precision(ma);
    cout << "\nincrease of working precision to " << ma << " bits.";
  }
  cout << "\n\nthis took " << used_time(T) << " seconds";
  }
  return 0;
  }
```

The following is a sample run of the program. Observe that we are more ambitious now and ask for the computation of the first 10^6 binary digits.

```
maximal error less than 2^{-m}; m = 1000000
a double value between 1/4 and 1, a =  0.56543254
iteration = 1   actual error is less than 2^L where L = -3
iteration = 2   actual error is less than 2^L where L = -9
iteration = 3   actual error is less than 2^L where L = -18
iteration = 4   actual error is less than 2^L where L = -18
increase of working precision to 32 bits.
iteration = 5   actual error is less than 2^L where L = -33
iteration = 6   actual error is less than 2^L where L = -33
increase of working precision to 64 bits.
iteration = 7   actual error is less than 2^L where L = -67
iteration = 8   actual error is less than 2^L where L = -67
increase of working precision to 128 bits.
iteration = 9   actual error is less than 2^L where L = -131
iteration = 10   actual error is less than 2^L where L = -131
increase of working precision to 256 bits.
iteration = 11   actual error is less than 2^L where L = -255
iteration = 12   actual error is less than 2^L where L = -255
increase of working precision to 512 bits.
```

```
iteration = 13   actual error is less than 2^L where L = -512
iteration = 14   actual error is less than 2^L where L = -512
increase of working precision to 1024 bits.
iteration = 15   actual error is less than 2^L where L = -1025
iteration = 16   actual error is less than 2^L where L = -1025
increase of working precision to 2048 bits.
iteration = 17   actual error is less than 2^L where L = -2048
iteration = 18   actual error is less than 2^L where L = -2048
increase of working precision to 4096 bits.
iteration = 19   actual error is less than 2^L where L = -4095
iteration = 20   actual error is less than 2^L where L = -4095
increase of working precision to 8192 bits.
iteration = 21   actual error is less than 2^L where L = -8191
iteration = 22   actual error is less than 2^L where L = -8191
increase of working precision to 16384 bits.
iteration = 23   actual error is less than 2^L where L = -16383
iteration = 24   actual error is less than 2^L where L = -16383
increase of working precision to 32768 bits.
iteration = 25   actual error is less than 2^L where L = -32770
iteration = 26   actual error is less than 2^L where L = -32770
increase of working precision to 65536 bits.
iteration = 27   actual error is less than 2^L where L = -65535
iteration = 28   actual error is less than 2^L where L = -65535
increase of working precision to 131072 bits.
iteration = 29   actual error is less than 2^L where L = -131072
iteration = 30   actual error is less than 2^L where L = -131072
increase of working precision to 262144 bits.
iteration = 31   actual error is less than 2^L where L = -262145
iteration = 32   actual error is less than 2^L where L = -262145
increase of working precision to 524288 bits.
iteration = 33   actual error is less than 2^L where L = -524289
iteration = 34   actual error is less than 2^L where L = -524289
increase of working precision to 1000008 bits.
iteration = 35   actual error is less than 2^L where L = -1000007
iteration = 36   actual error is less than 2^L where L = -1000007
this took 39.42 seconds
```

We see that we do two iterations for each precision. Why two? The first iteration brings the iterate close enough to $\sqrt{a}$ and in the second iteration the old and the new iterate are both close enough to $\sqrt{a}$ to trigger termination. The fact that we need one additional iteration in order to detect termination is no problem in the non-adaptive version of Newton's method. It is a problem for the adaptive version because it brings about one additional iteration for each fixed precision. Thus it effectively doubles the number of iterations.

The problem can be solved by a more careful analysis (which is actually simpler of what we have done above) and a modified algorithm. Assume inductively that we have an approximation $\widehat{x}_k$ with $d_k = |\sqrt{a} - \widehat{x}_k| \le 2^{-m_k}$. For $k = 0$, we set $\widehat{x}_0 = 1$ and have $d_0 \le 1/2$. Thus we can use $m_0 = 1$.

For the computation of $\widehat{x}_{k+1}$ we use mantissa length $2m_k + 2$. Thus $\epsilon = 2^{-2m_k-2}$. Then

$$\begin{aligned}
d_{k+1} &= |\widehat{x}_{k+1} - \sqrt{a}| \\
&\le |\widehat{x}_{k+1} - x_{k+1}| + |x_{k+1} - \sqrt{a}| \\
&\le |\widehat{x}_{k+1} - x_{k+1}| + |x_k - \sqrt{a}|^2 \\
&\le (10/4)\epsilon + d_k^2 \\
&\le 2^{-2m_k} + 2^{-2m_k} = 2^{-2m_k+1}
\end{aligned}$$

We may therefore set $m_{k+1} = 2m_k - 1$. OUCH, this does not quite work for the first iteration. Observe that $m_1 = 1 = m_0$ and hence we are not going to improve. We need to start with an iteration which guarantees $m_0 = 2$. We start with $x = 1$ if $\sqrt{a} \ge 3/4$ and with $x = 3/4$ otherwise. Our considerations lead to the following program.

⟨*Improved_Improved_Newton_demo.c*⟩≡

```
#include <LEDA/bigfloat.h>
main(){
while (true)
{ int m = read_int("\n\n\nmaximal error less than 2^{-m}; m = ");
double a = read_real("\na double value between 1/4 and 1, a =  ");
float T = used_time();
bigfloat xold;
bigfloat xnew;
if ( a >= 0.75 ) xnew = 1; else xnew = 0.75;
int k = 0;
int mk = 2;                        // abs(xnew - sqrt{a}) <= 2^{-mk}
bigfloat A(a);
do
{ mk = leda_min(2*mk - 1,m);
  bigfloat::set_precision(mk + 3);
  xold = xnew;
  xnew = (xold + A/xold)/2;
  k++;
  bigfloat::set_rounding_mode(TO_INF);
  bigfloat::set_precision(4*mk);
  bigfloat est = 2.0 * abs(xnew*xnew - A);
```

```
    cout << "\n\niteration = " << k << "  claimed error is at ";
    cout << "most 2^-mk where mk= " << mk;
    if ( est == 0 ) cout << "\n    actual error is zero";
    else cout << "\n    actual error is less than 2^L where L = " << ilog2(est);
    bigfloat::set_rounding_mode(TO_NEAREST);
    bigfloat::set_precision(mk+3);
  } while ( mk < m );

  cout << "\n\nthis took " << used_time(T) << " seconds";

  }

  return 0;
  }
```

And again the output of a sample run.

```
maximal error less than 2^{-m}; m = 1000000

a double value between 1/4 and 1, a =  0.56543254

iteration = 1  claimed error is at most 2^-mk where mk= 3
    actual error is less than 2^L where L = -7

iteration = 2  claimed error is at most 2^-mk where mk= 5
    actual error is less than 2^L where L = -7

iteration = 3  claimed error is at most 2^-mk where mk= 9
    actual error is less than 2^L where L = -18

iteration = 4  claimed error is at most 2^-mk where mk= 17
    actual error is less than 2^L where L = -18

iteration = 5  claimed error is at most 2^-mk where mk= 33
    actual error is less than 2^L where L = -35

iteration = 6  claimed error is at most 2^-mk where mk= 65
    actual error is less than 2^L where L = -67

iteration = 7  claimed error is at most 2^-mk where mk= 129
    actual error is less than 2^L where L = -131

iteration = 8  claimed error is at most 2^-mk where mk= 257
    actual error is less than 2^L where L = -260

iteration = 9  claimed error is at most 2^-mk where mk= 513
    actual error is less than 2^L where L = -515

iteration = 10  claimed error is at most 2^-mk where mk= 1025
    actual error is less than 2^L where L = -1027

iteration = 11  claimed error is at most 2^-mk where mk= 2049
    actual error is less than 2^L where L = -2051

iteration = 12  claimed error is at most 2^-mk where mk= 4097
    actual error is less than 2^L where L = -4099

iteration = 13  claimed error is at most 2^-mk where mk= 8193
    actual error is less than 2^L where L = -8198
```

```
iteration = 14  claimed error is at most 2^-mk where mk= 16385
    actual error is less than 2^L where L = -16388
iteration = 15  claimed error is at most 2^-mk where mk= 32769
    actual error is less than 2^L where L = -32773
iteration = 16  claimed error is at most 2^-mk where mk= 65537
    actual error is less than 2^L where L = -65538
iteration = 17  claimed error is at most 2^-mk where mk= 131073
    actual error is less than 2^L where L = -131074
iteration = 18  claimed error is at most 2^-mk where mk= 262145
    actual error is less than 2^L where L = -262147
iteration = 19  claimed error is at most 2^-mk where mk= 524289
    actual error is less than 2^L where L = -524295
iteration = 20  claimed error is at most 2^-mk where mk= 1000000
    actual error is less than 2^L where L = -1000003
this took 9.81 seconds
```

And finally the running time without the computation of the actual error.

⟨*Improved_Improved_Newton_demo2.c*⟩≡

```
#include <LEDA/bigfloat.h>
main(){
while (true)
{ int m = read_int("\n\n\nmaximal error less than 2^{-m}; m = ");
double a = 0.56545254; //read_real("\na double value between 1/4 and 1, a =  ");
cout << "\nwe compute root of " << a;
float T = used_time();
bigfloat xold;
bigfloat xnew;
if ( a >= 0.75 ) xnew = 1; else xnew = 0.75;
int k = 0;
int mk = 2;                       // abs(xnew - sqrt{a}) <= 2^{-mk}
bigfloat A(a);
do
{ mk = leda_min(2*mk - 1,m);
  bigfloat::set_precision(mk + 3);
  xold = xnew;
  xnew = (xold + A/xold)/2;
} while ( mk < m );
cout << "\n\nthis took " << used_time(T) << " seconds";
```

```
}
return 0;
}
```

The output is

```
maximal error less than 2^{-m}; m = 1000000
we compute root of 0.565453
this took 8.61 seconds
```

6 Newton Iteration with Floating Point Arithmetic

I leave it to one (or more) of you to write this section. I would like to treat the following topics.

1. Extend the preceding section to Newton iteration in general. Is it also possible to extend the adaptive precision scheme. What do we need to know about the first iterate?

2. Uspensky's algorithm for root isolation can be combined with Newton's method to compute arbitrary precision approximations of real roots. We run Uspensky's method until we have in interval I containing a single root of P and no root of P' and P''. Then P is a convex function and hence Newton's method will converge from an arbitrary starting point. However, it will not necessarily converge quadratically.

   - Can we modify Uspensky's algorithm so that it delivers isolating intervals which guarantee quadratic convergence of Newton's method.
   - Or should we run Newton's method right from the beginning. We would only have a guarantee of linear convergence at the beginning. What does this mean for the adaptive scheme?

3. In the separation bound approach to sign determination, we need to approximate an expression with absolute error bounded by $B/2$ where $B = u(E)/u(E)^{2^k}$. What precision is required in a floating point computation to guarantee this error bound? What does this imply for the complexity of computing the approximation?