Prof. W. Kahan, Math. Dept. Univ. of Calif. at Berkeley

Ideally, calculations with rounded arithmetic and uncertain data should be augmented by auxiliary calculations designed to account for errors due to rounding and uncertainty inherited from data by final results. The simplest schemes use Interval Arithmetic, an idea based upon the allocation to every variable of an interval known to contain the variable's value. Then arithmetic operations upon variables are replaced by the same operations upon intervals (actually, only endpoints of intervals need be manipulated), whence results stated as intervals assuredly enclose the desired results. Used naively, Interval Arithmetic can deliver awfully pessimistic (excessively wide) final intervals; to obtain better (narrower but still encompassing) intervals requires considerable skill. Alas, that is all entirely academic for the present because hardly any North American computer systems offer their users access to Interval Arithmetic. The idea is far more popular in Germany than in the USA despite having been devised here in the early 1960s. It's a familiar story; St. Matthew 14:57 ends some parables with the quotation " A prophet is not without honor, save in his own country, and in his own house."

This note describes, for four simple computations, how to manage after a fashion without Interval Arithmetic. Uncertainties in final results are computed by tracking errors and uncertainties through a computation, operation by operation, in a simple but tedious way. Let $\circledast$ denote a typical operation + , −, *, / .

First an estimate of the arithmetic's rounding error threshold ε is needed; we assume every *assignment* $x := y \circledast z$ actually stores a result $x = (y \circledast z)/(1 - \alpha)$ where α is unknown but $\varepsilon > |\alpha|$ is known. On correctly rounding machines, (over-)estimate ε thus:

$$\varepsilon := | ((4.0/3.0 \text{ rounded }) - 1.0)*3.0 - 1.0 | .$$

Can you see why this should work? Try it with a calculator first.

Second, we assume that uncertainties in data are known. If we wish to compute $X := Y \circledast Z$ but we know only y and z and error-bounds $eY > |Y-y|/\varepsilon$ and $eZ > |Z-z|/\varepsilon$, then we shall derive another error-bound $eX > |X-x|/\varepsilon$ from x, y, z, eY, eZ and ε . When $\circledast$ is + then $x = (y+z)/(1-\alpha)$ and $eX := eY + eZ + |x|$. But eX must be more complicated when $\circledast$ is * or / .

Consider first a sum $S_N := \sum_1^N B_j$ for which we have approximate data b_j and error-bounds $eB_j > |B_j - b_j|/\varepsilon$. We would compute $s_0 := 0$ and $s_N := s_{N-1} + b_N$ but get $s_N = (s_{N-1} + b_N)/(1 - \alpha_N)$, from which we infer $S_N - s_N = S_{N-1} - s_{N-1} + B_N - b_N - \alpha_N s_N$, and then $|S_N - s_N|/\varepsilon < eS_N := eS_{N-1} + eB_N + |s_N|$. We add this last assignment to the program that computes s_N to compute eS_N too, which is our *Running Error-Bound*. Roundoff in eS_N is ignored because it cannot materially affect a sum of positive terms unless N is huge, comparable with $1/\varepsilon$. Here is the complete program:

$s_0 := 0$; $eS_0 := 0$;
for N = 1, 2, 3, ... in turn, do
$s_N := s_{N-1} + b_N$;
$eS_N := eS_{N-1} + eB_N + |s_N|$.

In other words, we augment the recurrence programmed for S_N, but actually carried out upon s_N, with a recurrence carried out simultaneously upon eS_N to get a *Running Error Bound* $eS_N\varepsilon$ for the absolute error $|S_N - s_N|$, ignoring roundoff in eS_N.

The next example computation is a scalar product $S_N := \sum_1^N B_j C_j$ for which the data given is b_j, c_j, $eB_j > |B_j - b_j|/\varepsilon$ and $eC_j > |C_j - c_j|/\varepsilon$, and we seek to compute an approximate sum s_N and a bound $eS_N\varepsilon$ for its absolute error. Here is what to do:

$s_0 := 0$; $eS_0 := 0$;
for $N = 1, 2, 3, \ldots$ in turn, do
$p_N := c_N * b_N$; $s_N := s_{N-1} + p_N$;
$eS_N := eS_{N-1} + |p_N| + |s_N| + |c_N| * eB_N + (|b_N| + 3\varepsilon * eB_N) * eC_N$.

Now $eS_N > |S_N - s_N|/\varepsilon$, including the effects of roundoff in
$p_N = c_N b_N/(1 + \beta_N)$ and
$s_N = (s_{N-1} + p_N)/(1 - \alpha_N)$, where
$|\beta_N| < \varepsilon$ and $|\alpha_N| < \varepsilon$,
but ignoring roundoff in the computation of eS_N itself. Note that subscripted variables (arrays) would not normally be used for p_N, s_N and eS_N in practice.

The next example is a polynomial
$P_N(X) := A_0X^N + A_1X^{N-1} + \ldots + A_{N-1}X + A_N$.
computed from a recurrence
$P_N := A_N + X P_{N-1}$ starting with $P_0 := A_0$.
With the same notational conventions as before, the augmented recurrence for p_N and eP_N becomes

$p_0 := a_0$; $eP_0 := eA_0$;
for $N = 1, 2, 3, \ldots$ in turn, do
$t := x * p_{N-1}$; $p_N := a_N + t$;
$eP_N := eA_N + eX * |p_{N-1}| + |t| + |p_N| + (|x| + \varepsilon * eX) * eP_{N-1}$.

At the end, p_N approximates P_N and $eP_N > |P_N - p_N|/\varepsilon$. Later another simpler way will be presented, together with a recurrence and error-bound for the polynomial's derivative, all applied to the problem of finding zeros of a polynomial.

The final example is a continued fraction
$F_0 := A_0 + B_0/(A_1 + B_1/(A_2 + B_2/(A_3 + \ldots + B_M/F_{M+1} \ldots)))$.
The augmented recurrence is valid only if no denominator f_{N+1} is ever smaller than its uncertainty $eF_{N+1}\varepsilon$. Otherwise an entirely different scheme, too complicated to describe here, must be used to account for the possibility that $F_N = \infty$ but F_{N-1} is finite.

For $N = M, M-1, \ldots, 3, 2, 1, 0$ in turn, do
$q := b_N/f_{N+1}$; $f_N := a_N + q$;
$eF_N := eA_N + |q| + |f_N|$
$+ (|b_N| * eF_{N+1} + eB_N * |f_{N+1}|)/(|f_{N+1}| * (|f_{N+1}| - eF_{N+1}\varepsilon))$.

At the end, $eF_0 > |F_0 - f_0|/\varepsilon$.

Roundoff in Polynomial Evaluation

W. Kahan
Mathematics Dept.
Univ. of Calif.
Berkeley
Nov. 18, 1986

This note discusses two ways to assess the effects of rounding errors that occur during the calculation of a polynomial and its derivative in floating-point arithmetic. One way computes bounds for those effects during the computation of the polynomial and its derivative; the second way compares the rounding errors with hypothetical perturbations in the coefficients of the polynomial. Both ways have their uses, especially when the errors in computed zeros of a polynomial have to be assessed.

Introduction:
Given the coefficients a_j of the polynomial $A(x) = \sum_0^N a_j x^{N-j}$, and a numerical value z, we can compute both $p := A(z)$ and the derivative $q := A'(z)$ by means of ...
Horner's recurrence:

$q := 0$; $p := a_0$;
for $j = 1$ to N do { $q := z\,q + p$;
$p := z\,p + a_j$ } .

To demonstrate the validity of the recurrence, we need merely assign subscripts to the successive computed values thus:

$q_{-1} := 0$; $p_0 := a_0$; ... and $p_{-1} := 0$;
for $j = 1$ to N do { $q_{j-1} := z\,q_{j-2} + p_{j-1}$;
$p_j := z\,p_{j-1} + a_j$ } .

Then, by substituting for a_j, we find for all x that

$$A(x) = p_N + (x-z)\left(q_{N-1} + (x-z)\sum_0^{N-2} q_j x^{N-2-j} \right) ,$$

whence it soon follows that the final values of p and q are $p_N = A(z)$ and $q_{N-1} = A'(z)$ respectively. But no account has yet been taken of roundoff, to which this note is devoted.

We presume that, in any arithmetic operation " $x := y \circledast z$ ", the value actually computed is $x = (y \circledast z)(1+\xi)$ where the Greek letter ξ stands for a small rounding error about which we know only that $\varepsilon > |\xi|$; here ε denotes the relative uncertainty due to roundoff in the computer's floating-point arithmetic. We must introduce similar Greek letters to stand for every rounding error committed during Horner's recurrence, after which we find that the computed values p and q actually satisfy the following
perturbed recurrence:

$p_{-1} = q_{-1} = 0$; $p_0 = a_0$; ... and $\pi_0 = \kappa_0 = 0$;
for $j = 1$ to N { $q_{j-1} = (z\,q_{j-2}(1+\eta_{j-2}) + p_{j-1})/(1+\kappa_{j-1})$;
$p_j = (z\,p_{j-1}(1+\zeta_{j-1}) + a_j)/(1+\pi_j)$ } ;
... and $\eta_{N-1} = \zeta_N = 0$.

At this point we may either compute an upper bound for the effect of the perturbations upon the recurrence, or we may treat those perturbations as if they were equivalent to perturbations in the coefficients instead. The first approach is technically more intricate but philosophically simpler; let's try it first.

Computed Bounds upon Roundoff's Effect:
From the perturbed recurrence, substitute for a_j in the definition of $A(x)$ to get, for all x,

$$A(x) = p_N + \sum_0^N(\pi_j x - \zeta_j z)p_j x^{N-1-j} + (x-z)\left(q_{N-1} + \sum_0^{N-1}(\kappa_j x - \eta_j z)q_j x^{N-2-j} + (x-z)\sum_0^{N-2} q_j x^{N-2-j} \right) ,$$

whence it soon follows that

$$A(z) = p_N + \sum_0^N(\pi_j - \zeta_j)p_j z^{N-j} \quad \text{and}$$

$$A'(z) = q_{N-1} + \sum_0^{N-1}\left((\kappa_j - \eta_j)q_j + ((N-j)\pi_j - (N-1-j)\zeta_j)p_j \right) z^{N-1-j} .$$

Since no rounding error appears more than once in each of these formulas, the nonzero Greek letters can be replaced by $\pm\varepsilon$ to get best-possible bounds for the accumulated effect of roundoff:

$$|A(z) - p_N|/\varepsilon < |p_N| + 2\sum_1^{N-1}|p_j| r^{N-j} + |p_0| r^N \quad \text{where } r := |z| ;$$

$$|A'(z) - q_{N-1}|/\varepsilon < |q_{N-1}| + 2\sum_1^{N-2}|q_j| r^{N-1-j} + |q_0| r^{N-1} + \sum_1^{N-1}(2N-2j-1)|p_j| r^{N-1-j} + (N-1)|p_0| r^{N-1} .$$

The right-hand sides of these inequalities are polynomials in r with coefficients derived from $|p_j|$ and $|q_j|$, so they can be computed by recurrence too. To do that, here is an

augmented recurrence:

```
r := |z| ;  q := 0 ;  p := a_0 ;  e := |p| ;  d := -e/r ;
for  j = 1 to N  do  (  q := z q + p ;
                        d := r d + e + |q+q| - |p| ;
                        p := z p + a_j ;
                        e := r e + |p+p|  ) ;
e := e - |p| ;  d := d - |q| .
```

Now $|A(z) - p|/\varepsilon < e$ and $|A'(z) - q|/\varepsilon < d$ except for over/-underflow and ignorable roundoff incurred during the calculation of e and d. Verifying that the last two inequalities do follow from the previous two is a challenging exercise in algebraic manipulation; that verification will confirm that the two sides of each inequality could approach each other arbitrarily closely in the event, albeit unlikely, that all the rounding errors had magnitudes ε and appropriate signs.

The augmented recurrence is most useful during the computation of a zero of $A(x)$. For instance, if z is approximately a zero of A then the error in z is approximately $-A(z)/A'(z)$, the next step of Newton's iteration. We may infer from the recurrence that $|A(z)| < |p| + e\varepsilon$ and that $|A'(z)| > |q| - d\varepsilon$, whence follows $|-A(z)/A'(z)| < (|p| + e\varepsilon)/(|q| - d\varepsilon)$ provided this is positive; otherwise roundoff so obscures A' that no such error bound for z can be estimated.

Another instance arises during an iteration to compute a zero of $A(x)$. That iteration has to be stopped when z is so close to a zero that roundoff makes further iteration probably futile. A good time to stop is when $|p| < 2e\varepsilon$; this implies that $|A(z)|$ cannot be much bigger than the roundoff that accrued during its computation. (The factor 2 is necessary to ensure that some machine-representable argument z exists so close to a zero of A that such an inequality can be satisfied. Without that factor, there would be some risk that $|p| \geq e\varepsilon$ for all arguments z, even those adjacent to a zero of A. On the other hand, the factor 2 is big enough because $e\varepsilon$ exceeds the change in $A(z)$ that would be caused by changing z to one of its neighbors.)

During iteration to find a zero, the bound $d\varepsilon$ upon the error in $A'(z)$ is not very useful because substantial errors in $A'(z)$

cause the iteration to misbehave in ways that are usually easy to recognize without d . Therefore, during the iteration, d can be omitted from the augmented recurrence, which then simplifies to a short form that runs significantly faster when N is large:

```
r := |z| ;  q := 0 ;  p := a_0 ;  e := |p|/2 ;
for  j = 1 to N  do  {  q := z q + p ;
                        p := z p + a_j ;
                        e := r e + |p|  } ;
e := e - |p| + e .
```

After the iteration has terminated and z has been accepted as an approximate zero, running the previous augmented recurrence once provides an *estimate* $(|p|+e\varepsilon)/(|q|-d\varepsilon)$ of a bound upon the error in z . That estimate is *not* a rigorous bound; it was based upon an estimate $-A(z)/A'(z)$ that could *underestimate* the error in z . How badly? At worst by a factor 1/N according to

Laguerre's Theorem: The polynomial A(x) of degree N must have a zero ζ satisfying $|\zeta - z + A(z)/A'(z)| \le (N-1)|A(z)/A'(z)|$.

Proof: We know that $A(x) = a_0(x-x_1)(x-x_2)(\ldots)(x-x_{N-1})(x-x_N)$, where $x_1, x_2, \ldots, x_{N-1}, x_N$ are the (unknown) zeros, real and complex, of A . Let ζ be whichever of them is closest to z . Since $\zeta - z + A(z)/A'(z) = (1 - \sum_j (z-\zeta)/(z-x_j)\ )\ A(z)/A'(z)$, cancel " 1 " and take magnitudes to deduce the theorem.

Therefore the error in an approximate zero z cannot exceed

$$N\,(|p|+e\varepsilon)/(|q|-d\varepsilon) .$$

This error bound is rigorous but, like Laguerre's theorem, it is usually pessimistic by a factor near N which may be annoying if N is very big. A rigorous error bound that is usually far less pessimistic costs slightly more computation and much more thought.

The simplest thoughts arise when z approximates a real zero of a real polynomial A(x) . The sign of A(z) is the same as that of $p+e\varepsilon$ and $p-e\varepsilon$ when they have the same signs; otherwise the sign of A(z) is obscured by roundoff, as it usually would be when z is the best available approximation to a zero of A(x) . Now let η be any approximate bound for the error in z ; for instance, try $\eta := (|p|+e\varepsilon)/|q|$. We can check whether η truly bounds the error in z by running the short form of the augmented recurrence twice to see whether the signs of $A(z-\eta)$ and $A(z+\eta)$ are opposite, taking roundoff into account. Usually those signs do differ, and then we know for sure that A(x) vanishes between $z-\eta$ and $z+\eta$. Otherwise accept what Laguerre's theorem tells us.

When complex zeros of a polynomial are being computed, error bounds better than are provided by Laguerre's theorem require a lot of thought. Because complex variables lie beyond the syllabus of this course, only a rough outline of those thoughts will be sketched here. They begin with the divided difference

$$\begin{aligned} \Delta f(\{x, y\}) &:= (f(x) - f(y))/(x - y) \quad \text{if } x \ne y , \\ &:= f'(y) \quad \text{if } x = y . \end{aligned}$$

It resembles the derivative in many ways besides their similar definitions; they also figure in similar estimates for the zeros of a function. For instance, ...

Lemma: Let z be fixed at the center of some region $|x-z| \le \delta$ throughout which $f(x)$ is continuously differentiable, and suppose also that $|\Delta f(\langle x, z\rangle)| > |f(z)|/\delta$ at every x therein. Then $f(x)$ must vanish *at least* once inside that region. (It may be an interval on the real axis or a disk in the complex plane. And if also $|\Delta f(\langle x, y\rangle)| > 0$ at all x and y in the region, $f(x)$ vanishes *just* once therein.)

Proof: Construct the map $\Phi(x) := x - f(x)/\Delta f(\langle x, z\rangle)$. This map is inspired by Newton's and the Secant iterations. Since the divisor cannot vanish, Φ must be continuous throughout the region. Moreover, $\Phi(x)-z = -f(z)/\Delta f(\langle x, z\rangle)$, which implies $|\Phi(x)-z| < \delta$ throughout the region, which means that Φ is a continuous map of this closed bounded convex region into itself. By Brouwer's fixed-point theorem, Φ must have at least one fixed point $x = \Phi(x)$ in the region; that is where $f(x) = 0$.

The derivative and the divided difference of our polynomial $A(x)$ share another property; they can be computed without any division from a revised version of Horner's recurrence as follows:

$$Q := 0 \; ; \quad p := a_0 \; ;$$
$$\text{for } j = 1 \text{ to } N \text{ do } \langle \; Q := y\,Q + p \; ;$$
$$p := z\,p + a_j \; \rangle \; .$$

The final values of $p = A(z)$ and $Q = \Delta A(\langle y, z\rangle)$ would be correct but for rounding errors which shall be ignored here to simplify the exposition. How does Q vary as y runs about some tiny region containing a zero of $A(x)$ close to z ? If $y = z$, $Q = q = Q'(z)$ as computed by Horner's recurrence. Otherwise, using the subscripted values p_j we introduced to explain that recurrence, we can infer that

$$Q - q := (y-z) \sum_{j=0}^{N-2} p_j S_{N-1-j} \quad \text{where}$$
$$S_k := (z^k - y^k)/(z-y) = \sum_{j=0}^{k-1} z^j\, y^{k-1-j} \; .$$

If we can find some $s > \max(|z|, |y|)$, then $|S_k| < k\,s^{k-1}$ and

$$|Q-q| < |y-z| \sum_{j=0}^{N-2} (N-1-j)|p_j| s^{N-2-j} \; ,$$
$$= |y-z|\, R(s)$$

where $R(s)$ in the last inequality is a polynomial in s that can be computed by another augmented recurrence similar to the one that computes d above. To determine s , we choose any known error bound $\xi := N(|p|+e\varepsilon)/(|q|-d\varepsilon)$, say, and assume that $|y-z| < \xi$; then $s := |z| + \xi$. Then we run another augmented recurrence to evaluate $R(s)$, and deduce that for all $|y-z| < \xi$

$$|\Delta A(\langle y,z\rangle)| = |q + (Q-q)| \ge |q| - |Q-q| > |q| - \xi\, R(s)$$

to within a small computable allowance for roundoff. If the last expression exceeds $(|p|+e\varepsilon)/\xi$, as it usually does, then the Lemma tells us that $A(x)$ vanishes somewhere in the region

$$|x-z| < (|p|+e\varepsilon)/(|q| - \xi\, R(s)) \; ,$$

which is usually much tinier than ξ .

A number of details have been omitted from the foregoing account because they lie beyond the scope of this course. The important conclusions to be drawn from what has been presented so far are that we can compute a zero of a polynomial as accurately as we like if we carry enough precision, and that we can prove our result correct taking roundoff into account with complete rigor. Although it is possible to compute better error bounds that come ever closer to the limits of uncertainty imposed by roundoff, such bounds are hardly ever worth their cost because other sources of uncertainty so often predominate over roundoff.

Interpreting Roundoff as Perturbations in Data:
Let us return to the perturbed recurrence and express p_j in terms of the rounding errors (the Greek letters) and the given coefficients a_j. Instead of computing $A(z) := \sum_0^N a_j z^{N-j}$, we can prove by induction that actually $p_N = A(z) := \sum_0^N a_j z^{N-j}$ where

$$a_j = a_j(1+\zeta_j)(1+\zeta_{j+1})(\ldots)(1+\zeta_{N-1})/((1+\pi_j)(1+\pi_{j+1})(\ldots)(1+\pi_N))$$
$$= a_j(1\pm\varepsilon)^{\pm(2N-2j+1)} \quad \text{if } j \neq 0, \text{ otherwise } a_0(1\pm\varepsilon)^{\pm 2N}.$$

In other words, the computed value p, obtained instead of the desired value $A(z)$, is exactly what would have been computed without roundoff if each coefficient a_j had first been perturbed to a nearly indistinguishable number $\mathcal{a}_j$. This is the sense in which roundoff committed during the computation of $A(z)$ is no worse than a few rounding errors per coefficient committed before that computation. If the given coefficients are uncorrelatedly uncertain by as much as $2N$ units in their last significant digits, then that uncertainty in $A(z)$ will dominate whatever uncertainty subsequently accrues to p because of roundoff. This view of the rounding errors is called a *"Backward Error-Analysis"*.

Let us reconsider the computation of a zero of $A(x)$ from this point of view. We shall accept z as a purported approximation to a zero of A when p, the computed value of $A(z)$, is deemed negligible; but that will actually mean that the perturbed polynomial $p = \mathcal{A}(z)$ is negligible, so z will actually lie close to a zero of the perturbed polynomial $\mathcal{A}$. Because the zeros of a polynomial are known to be continuous functions of its coefficients, and because the coefficients of A and $\mathcal{A}$ are so nearly indistinguishable, one might hope their zeros are almost the same too. However, some closely neighboring polynomials A and $\mathcal{A}$ have surprisingly different zeros. For example, let

$$A(x) := x^{12} - 12x^{11} + 66x^{10} - 220x^9 + 465x^8 - 792x^7 + 924x^6 - 792x^5 + 495x^4 - 220x^3 + 66x^2 - 12x + 1$$
$$= (x-1)^{12},$$

and let $\mathcal{A}(x) := A(x) - x^6/10^6$. In other words, the perturbed polynomial $\mathcal{A}(x)$ is obtained from $A(x)$ by replacing $924x^6$ by $923.999999x^6$, a change in the ninth significant decimal. Then, although all twelve zeros of A are at $z = 1$, two of the zeros of $\mathcal{A}$ are at $z = 0.729843788$ and $z = 1.370156212$. (These are the zeros of $(x-1)^2 - x/10$, which you should confirm as a divisor of $\mathcal{A}(x)$.) A similar but slightly more extreme example is constructed from the same $A(x) := (x-1)^{12}$, but now perturb it to $\mathcal{A}(x) := A(x) - A(-x)/5_{10}9$, changing each coefficient in its tenth sig. dec. or beyond. For instance, $12x$ gets changed into $12.0000000024x$, and $924x^6$ into $923.9999998152x^6$. You should be able easily to compute the two real zeros $z = 1.36828744$ and $z = 0.73084059$ of $\mathcal{A}(x)$.

The foregoing two examples may give the false impression that the zeros of a polynomial can be hypersensitive to tiny perturbations in its coefficients only if the zeros are repeated. Actually the truth is slightly but crucially different from that. Zeros can be hypersensitive to tiny perturbations in coefficients only if such tiny perturbations could cause the zeros in question to change their multiplicities. An explanation and proof of this assertion would be too complicated to include in this note; instead, the assertion will be illustrated by an example. This example is similar to one discovered in the late 1950's by James H. Wilkinson and used for the same purpose. Let

$$A(x) := x^{12} - 78x^{11} + 2717x^{10} - 55770x^9 + 749463x^8 - 6926634x^7 + 44990231x^6 - 206070150x^5 + 657206836x^4 - 1414014888x^3 + 1931559552x^2 - 1486442880x + 479001600$$
$$= (x-1)(x-2)(x-3)(\ldots)(x-10)(x-11)(x-12)$$
$$= \Gamma(x)/\Gamma(x-12) = (x-1)!/(x-13)! .$$

Its zeros, the consecutive integers from 1 to 12, do not seem especially close together, but in fact an extremely tiny change to its coefficients can change the zeros enough to make two of them coalesce. Specifically, $A(x) := A(x) - \lambda A(-x)$ has a double zero at $z = 8.4835138$ when $\lambda = 5.600278_{10}-10$. Although the zeros of A are hypersensitive to roundoff, these numbers z and λ can be calculated easily on a programmable calculator by means interesting enough to merit inclusion in this note.

The double zero z of $A(x)$ must satisfy $A(z) = A'(z) = 0$; that means $A(z) - \lambda A(-z) = A'(z) + \lambda A'(-z) = 0$. Eliminating λ produces an equation, $A'(z)/A(z) + A'(-z)/A(-z) = 0$, that identifies z as one of the 22 finite zeros of

$$A'(x)/A(x) + A'(-x)/A(-x) = \sum_{j=1}^{12} (1/(x-j) - 1/(x+j)) .$$

Every such zero lies between two consecutive nonzero integers between -12 and 12, so it is easy to compute accurately by Newton's or Secant iteration. Each such zero z determines a corresponding $\lambda = A(z)/A(-z) = \Gamma(z)\,\Gamma(z+1)/(\Gamma(z-12)\,\Gamma(z+13))$. The smallest λ and its z are the ones exhibited above.

The foregoing examples warn us that some polynomials are so hypersensitive to roundoff in their coefficients that their zeros cannot be determined without carrying extravagant precision that may be unjustified if the coefficients are intrinsically uncertain by as little as a few rounding errors. In such cases, we should try to find out where the polynomial came from; it may have come from a problem that determined its roots quite accurately until it was transformed into an explicit polynomial equation. For example $A(x) = (1-\lambda)x^{12} - 78(1+\lambda)x^{11} + 2717(1-\lambda)x^{10} - \ldots + 479001600(1-\lambda)$ has the same zeros as the expression

$$\Gamma(x)\,\Gamma(x+1)/(\Gamma(x-12)\,\Gamma(x+13)) - \lambda ,$$

but, for any fixed tiny value of λ, at least about $-\log(|\lambda|)$ more sig. dec. must be carried when the polynomial $A(x)$ is used than when the latter expression is used to calculate those zeros to any preassigned accuracy, as we shall see.

Condition Numbers:

How sensitive can a zero z of some function $f(x)$ be to small perturbations Δf in f? *Condition numbers* answer questions like this in a quantitative way. Before we compute a condition number, we must choose a *norm* $\|\ldots\|$ to measure the size of small perturbations; ideally $\|\Delta f\|$ takes the same value for all perturbations Δf that are about equally likely or about equally (in)significant. There is no reason why $\|\Delta f\|$ should not depend upon z; hence, one such norm for perturbations

$$\Delta A(x) = \sum_0^N \Delta a_j\, x^{N-j}$$

in the polynomial $A(x)$ might be

$$\|\Delta A\| := \sum_0^N |\Delta a_j\, z^{N-j}| ,$$

which serves as an upper bound for $|\Delta A(z)|$ (though not for $|\Delta A(x)|$ when $|x| > |z|$). Given a norm for Δf, a *relative* condition number for the infinitesimal changes Δz caused by infinitesimal perturbations Δf in f would be

$$\kappa(z,\ f,\ \|\ldots\|) \ := \ \max_{\Delta f \neq 0} |\Delta z/z| / (\|\Delta f\|/\|f\|)$$

where the maximum is taken over all infinitesimal nonzero Δf and Δz is obtained from a root of $(f+\Delta f)(z+\Delta z) = 0$ closest to the root z of $f(z) = 0$. Consequently, $\Delta z = -\Delta f(z)/f'(z)$ if $f'(z) \neq 0$ and terms of order $(\Delta z)^2$ are ignored, and then

$$\kappa(z,\ f,\ \|\ldots\|) \ = \ \left(\max_{\Delta f \neq 0} |\Delta f(z)| / \|\Delta f\|\right) \|f\| / |z\, f'(z)| \ .$$

The maximum in parentheses depends upon z and $\|\ldots\|$ but not upon f, and turns out to be $\|e(z)^*\|$ where $e(x)^*$ is called "the evaluation functional" because it extracts a value $f(x) = e(x)^* f$ from any function f. For the example $\|\Delta A\|$ above,

$$\|e(z)^*\| \ = \ \max_{\Delta A \neq 0} |\Delta A(z)| / \|\Delta A\| \ = \ 1 \ , \ \text{so}$$

$$\kappa(z,\ A,\ \|\ldots\|) \ = \ \|A\| / |z\, A'(z)| \ .$$

This condition number κ roughly bounds the magnification ratio $(|\Delta z|/|z|)/(\|\Delta A\|/\|A\|)$ for relatively tiny perturbations ΔA.

A similar analysis applies to equations $f(x) = \lambda$ when $f(x)$ can be computed correct to nearly as many sig. dec. as are carried by the arithmetic. The appropriate norm then is actually a semi-norm $\|\Delta f\| = |\Delta f(z)|$, and the appropriate relative condition number is

$$\kappa(z,\ f-\lambda,\ \|..\|) \ := \ \max_{\Delta f} (|\Delta z|/|z|)/(\|\Delta f\|/\|f\|) \ = \ |\lambda|/|z\, f'(z)| \ .$$

Let us compare condition numbers for the foregoing examples

$$A(x) \ := \ (1-\lambda)x^{12} - 78(1+\lambda)x^{11} + 2717(1-\lambda)x^{10} - \ \ldots \ + 479001600(1-\lambda)$$

and

$$f(x) \ := \ \Gamma(x)\,\Gamma(x+1) / \left(\Gamma(x-12)\,\Gamma(x+13)\right) \ = \ A(x) / \left(\Gamma(x+13)/\Gamma(x+1)\right) + \lambda \ .$$

Since $\|A\| / \left(\Gamma(z+13)/\Gamma(z+1)\right)$ lies between $1 \pm \lambda$ we soon find that

$$\kappa(z,\ A,\ \|\ldots\|) / \kappa(z,\ f-\lambda,\ \|..\|) \ = \ (1 \pm \lambda)/\lambda \ ,$$

which vindicates the earlier claim that about $-\log_{10}(|\lambda|)$ more sig. dec. must be carried to solve $A(z) = 0$ than to solve $f(z) = \lambda$ equally accurately. And when $\lambda = 0$ the condition number of the integer $z = 1,\ 2,\ \ldots,$ or 12 as a zero of A is

$$\kappa(z,\ A,\ \|\ldots\|) \ := \ (z+12)!/((12-z)!\,(z!)^2)$$
$$= \ 64664600 \quad \text{when} \quad z = 9 \ ,$$

which explains a loss of 7 sig. dec. when that zero is computed.

The foregoing condition numbers κ pertain only to simple zeros z; condition numbers for multiple zeros lie beyond this course.

Acknowledgements:
Although this note was prepared for an introductory class in Numerical Analysis, the computable error bounds are based upon researches supported at times by the U. S. Office of Naval Research under contract number N00014-76-C-0013, and by the Air Force Office of Scientific Research under AFOSR-84-0158.

ROOTS OF EQUATIONS, ZEROS OF FUNCTIONS

W. Kahan
Math. Dept., and EE & Comp. Sci. Dept.
University of California at Berkeley

The *roots* of the equation $f(x) = 0$ are the *zeros* of the function $f(x)$. For example, if $f(x) = (x-2)(x-3)$ then the roots of the polynomial equation $f(x) = 0$ are $x = 2$ and $x = 3$, the zeros of the polynomial $f(x)$. To call them "the roots of the polynomial" is an abuse of language because they might then be confused with the *square roots* of the polynomial, namely $\pm\sqrt{f(x)}$. These two square roots are the roots z of the polynomial (in z) equation $z^2 = f(x)$, not the roots x of the polynomial (in x) equation $f(x) = 0$.

Often an equation $F(x, y) = 0$ is said to be solvable for x as a function of y, or for y as a function of x, despite that no explicit expression for either is available. What this means is that, in principle, given a procedure that defines $F(x, y)$ and given a numerical value for y, there exists a program that will solve the equation $F(x, y) = 0$ for another numerical value x that can be calculated as accurately as you like if you let the program run long enough. Or vice-versa. Thus does the equation $F(x, y) = 0$ define x or y implicitly as a function of the other at the cost of solving numerically one equation in one real unknown. However, knowing or believing that a program exists is not the same as having the program in hand.

Canned Software:

One nonlinear equation in one real unknown is just the kind of equation solved automatically by the [SOLVE] key introduced in 1978 on the hp-34C calculator. Subsequent Hewlett-Packard calculators have improved versions of that capability. Other calculators may have capabilities comparable but less proficient in that either they require first guesses to straddle a desired zero, or they require unknowable parameters to tell when to stop searching, or both. Similar limitations are imposed in libraries available from companies like IMSL, NAG, C.ABACI, ... for many lines of computers, including IBM /370, DEC VAX, Sun Sparc, IBM PC, ... An equation solver is found in many spreadsheets and in systems like MathCad, Mathematica, Maple, Derive, ... ; Eureka contains a particularly well implemented solver. Some such software is accessible via the administrator of the Math. Department's computers, or the University's Computing Center, or the Engineering Library.

No "canned" equation solver can be foolproof; in fact, you should be able to demonstrate why an infallible general-purpose equation solver is as impossible as perpetual motion. That is why everyone, regardless of computing resources, will find a certain amount of theory useful. And everyone in the class needs practice using more than one general-purpose equation solver now to achieve a proficiency that will pay off later.

Iteration:

The most common way to solve an equation $f(x) = 0$ is via an *iteration* $x_{N+1} = h(x_N)$ intended to generate a sequence $x_0, x_1, x_2, \ldots$ of iterates x_N convergent to a desired root. Such an iteration is constructed from the equation $f(x) = 0$ by first transforming it into a suitable *equivalent* equation $x = h(x)$; these equations are equivalent just when they have the same roots. In other words, each zero z of f , where $f(z) = 0$, should be a *fixed point* $z = h(z)$ of h , and vice-versa. Moreover, h should be a *contraction map* in the sense that $h(x)$ and $h(y)$ are closer to each other than are x and y whenever x and y are both not too far from a desired fixed point z ; this is the gist of the *Lipschitz condition* $|h(x)-h(y)| / |x-y| < \lambda < 1$ that h should satisfy. The smaller is λ , the faster is convergence because then, as $N \to \infty$,

$$|x_{N+1}-z| = |h(x_N)-h(z)| < \lambda |x_N-z| < \ldots < \lambda^N |x_1-z| \to 0 .$$

Given an expression for the equation $f(x) = 0$, an equivalent equation of the form $x = h(x)$ is easy to construct in many ways. Not so easy is to get an h that satisfies a Lipschitz condition with a small λ or, in other words, that has a small derivative $|h'(x)| < \lambda$ for all x near the desired fixed point. And then there is the problem of finding an initial iterate x_0 so close to the desired fixed point that the Lipschitz condition will ensure the iteration's convergence. And whether iteration converges or not, it has to be stopped somehow; and then comes the problem of discovering how close the last iterate is to the desired fixed point. These are some of the problems considered in this course.

Overview of the Theory:

The archetypal example of a rapidly convergent iteration is Newton's, for which the iterating function is constructed thus:

$$h(x) := x - f'(x)^{-1} f(x) .$$

The construction applies not only to scalar equations, the topic covered in this course, but also to systems of equations wherein f and h are vector-valued functions of a vector argument x . To construct the vector $h(x)$ solve the system of *linear* equations

$$f'(x) (h(x) - x) + f(x) = 0$$

for the increment $h(x) - x$; here the derivative of f is the *Jacobian* matrix f' of first partial derivatives that figures in the differential relationship $f(x+dx) = f(x) + f'(x)\,dx$ valid for infinitesimal vector displacements dx . The desired zero z of f is a fixed point $z = h(z)$ of the iterating function h . Of course, the desired z is not guaranteed to exist.

One way to recognize whether a fixed point exists is the

Brouwer/Schauder fixed-point theorem:

If h maps a compact convex region continuously into itself then that region contains *at least one* fixed point $z = h(z)$. (In a finite-dimensional vector space, a *compact convex* region is one of finite breadth consisting of its boundary and all points lying on line segments joining pairs of boundary points.)

The iterating function h should ideally satisfy a Lipschitz condition (as defined above except that $|x-y|$ is taken to mean the length of the vector $x-y$) in which the Lipschitz constant λ is as small as possible. The advantage of Newton's iteration over other ways of defining h is that normally λ is very tiny for Newton's h. Regardless of how h may be constructed, it is a candidate for the application of the following

Contraction mapping principle: If h maps a closed region into itself and satisfies therein a Lipschitz condition with $\lambda < 1$, then that region contains *just one* fixed point of h and the iteration $x_{N+1} := h(x_N)$ converges to it from any x_0 in the region. (A *closed* region contains its own boundary.)

This principle is valuable in theory for discovering when and where to start iterating; it is also valuable in practice for deciding when to quit. The problem of locating *one* fixed point of h with adequate accuracy is provably solved when an adequately small region is found throughout which both $h(x)-x$ and $h'(x)$ are small enough for the contraction mapping principle to apply.

The ideas in the last three paragraphs belong to the syllabus of Math. 128 B . However, Math. 128 A does deal with two special cases:- systems of linear equations in many unknowns, and one nonlinear equation involving only one scalar unknown, usually a real variable. The syllabus of Math. 128 A includes easy proofs of the Brouwer/Schauder fixed-point theorem and the Contraction Mapping principle when $h(x)$ is a differentiable real function of one real variable x, in which case the regions in question turn out to be simply intervals on the real axis.

For example, you should be able to use the contraction mapping principle to verify that the equation $x = e^{-x}$ has just one root (not a larger odd number of them) between 0.5671 and 0.5672 .

The idea behind Newton's iteration figures also in a scheme that estimates the error in an approximation y so close to z, the desired zero, that $|f(y)|$ is tiny and $|f'(x)|$ relatively big at all x near enough to y ; more precisely, ...

Suppose $0 \le f(y) \le \Phi$ for some small Φ, and suppose some δ exists such that $f'(x) > \Phi/\delta$ throughout $y-\delta \le x \le y$.
Then $f(z) = 0$ at *just* one point z in $y-\delta < z \le y$.

Proof: If $f(x)$ vanished more than once between $y-\delta$ and y then Rolle's theorem would force $f'(x) = 0$ at least once between them, which would contradict $f'(x) > \Phi/\delta$. On the other hand, $f(y) - f(y-\delta) = \int_{y-\delta}^{y} f'(x)\,dx > (f(y)/\delta) \int_{y-\delta}^{y} dx = f(y)$, which implies $f(y-\delta) < 0 \le f(y)$; so $f(z) = 0$ between $y-\delta$ and y . (A similar result can be proved for a vector-valued function $f(x)$ of a vector x provided *"just"* is replaced by *"at least"* .)

δ must be determined by trial-and-error, initially rather bigger than $|\Phi/f'(y)|$, decreasing until δ is not much bigger than will barely satisfy the stated supposition. For example, try $f(x) = x - e^{-x}$, $y = 0.56715$, $\Phi = 0.00002$, and $\delta = 0.01$ initially, and deduce that $\delta < 0.00005$ finally. (Draw pictures!)

More Theory for One Equation in One Real Variable:

The theory of equations and iterations involving just one real variable includes several facts worth knowing in practice though their proofs may be so intricate as to lie beyond the syllabus of any undergraduate class. Here follow some of those facts:

Let us say that h *swaps* two points x and y just when
$$h(x) = y \neq x = h(y) .$$
An astonishing relation exists between swapping and convergence.

The no-swap condition: Suppose the real function h maps a closed interval continuously into itself, and so has at least one fixed point therein. Then the iteration $x_{N+1} := h(x_N)$ converges to a fixed point from every initial x_0 in that interval if and only if no two points of it are swapped by h .

This striking relation between swapping and convergence first turned up in the mid 1960's in the work of a Russian mathematician A. N. Sharkovsky. To illustrate its power, use it to produce a short proof of the contraction mapping principle on an interval. Another implication, discovered in 1977, is the weakest known condition sufficient (but not necesary) for the convergence of Newton's iteration, for which $h(x) = x - f(x)/f'(x)$. The condition is expressed in terms of *convexity* and *monotonicity*; a real function $g(x)$ is called "convex" if its second derivative $g''(x)$ is never negative, and is called "monotone non-decreasing" if its first derivative $g'(x)$ is never negative.

The Convexity condition: Suppose that f can be expressed throughout some interval as a difference $f(x) = u(x) - v(x)$ between two convex functions u and v , one monotone non-decreasing and the other monotone non-increasing. Then Newton's iteration $x_{N+1} := x_N - f(x_N)/f'(x_N)$ must, from every x_0 in that interval, either converge within it or escape from it but not meander about the interval forever.

This situation, which merely requires f' not to fluctuate too fast, is encountered often in practice. For instance, suppose $p(t)$ and $q(t)$ are two non-constant polynomials in t with all coefficients non-negative and with $p(0) > q(0)$. Then certain financial calculations concerning rates of return on investments could require the solution of one of the equations
$$p(1/x) - q(x) = 0 \quad \text{for } x > 0 , \quad \text{or}$$
$$\log(p(e^{-z})) - \log(q(e^{z})) = 0 \quad \text{for any real } z .$$
Aided by the convexity condition, you should verify that Newton's iteration will converge, from any positive initial iterate, to the root of whichever of these two equations it is applied to.

Although the convexity condition's validity is hard to prove in the full generality stated above, you should be able easily to prove a simpler version in which either u or v is presumed to be constant. (That simpler version can be found in some older texts, especially French, associated with the names "Fourier" and "Dandelin.") And you should be able to apply it to the following example, which is motivated by a problem that arose in the design of a transistor circuit. For every $x > 1$ the equation $y - \ln(y) = x$ has two roots y of which we wish to compute the

smaller as a function of x. Transform the equation into an equivalent one which may always be solved by Newton's iteration starting from an initial iterate $y_o = 0$. (Another scheme is faster, especially when high accuracy is required and x is very near 1, but is much harder to analyze;- apply Newton's iteration to the original equation $(x-y) + \ln(y) = 0$ starting from $y_o := \max\{ \exp(1-x-4/(1+\sqrt{(1+8/(x-1))})) , 1/(\exp(x)-1) \}$.)

When the graph of f is too undulatory, Newton's iteration may meander indefinitely unless it is started close enough to a zero of f. For example, try to solve $f(z) = 0$ when

$$f(x) = 3x^5 - 10x^3 + 23x$$ (Plot a graph!)

using Newton's iteration starting from any x_o between about 0.923 and 1.069. For another example, determine how close to π the initial x_o must be to ensure that Newton's iteration will converge from there to the zero of $f(x) = \arctan(x-\pi)$.

If Newton's iteration converges at all, it usually does so *quadratically*, which is very fast. A typical example is

$$f(x) = x/(1-x) ,$$

for which you should obtain formulas in closed form for the successive Newton iterates x_N in terms of x_o. An example of abnormally slow convergence is provided by $f(x) = x^{10}$, for which you should do the same again. But no matter how slowly the iterates x_N may converge to a zero of an analytic function $f(x)$, the successive function values $f(x_N) \to 0$ faster than some constant multiple of e^{-N} as $N \to \infty$. Consequently, unless the starting iterate is most unfortunate, Newton's iteration cannot take very long, if it will converge at all, to produce function values $f(x_N)$ that are tinier than their own rounding errors or else small enough to underflow, at which point we should stop. For example, choose any integer $m > 0$ and any positive initial iterate $x_o < m^{1/m}$ to solve the equation $\exp(-x^{-m}) = 0$ using Newton's iteration $x_{N+1} = x_N - x_N^{m+1}/m$; then observe how slowly $x_N \to 0$, the more so when m is big, and yet how quickly $\exp(-x_N^{-m}) \to 0$ regardless of m.

Newton's iteration may seem very special amongst iterations generally, but there is a perverse sense in which almost every iteration on one scalar variable can be regarded as Newton's. If the iteration $x_{N+1} := h(x_N)$ converges to a fixed point z from every x_o in a neighborhood of z within which $h(x)$ is continuous, then this iteration is actually Newton's applied to a differentiable function $f(x)$ that vanishes at $x = z$. In fact, $f(x) = (\pm\text{const.}) \exp(\int^x dt/(t-h(t)))$; it may have a singularity as well as a zero at z.

As zero-finding methods go, Newton's iteration is pretty good except when the derivative f' is expensive or impossible to calculate. If the cost of calculating f and f' together exceeds by more than about 44% the cost of calculating f alone then the Secant iteration may be more economical than Newton's.

The Secant Iteration:

The Secant iteration $x_{N+1} := H(x_N, x_{N-1})$, where

$$H(x, y) := x - f(x)/((f(x)-f(y))/(x-y)) ,$$

works with pairs of successive approximations to a zero of f , so Sharkovsky's no-swap condition cannot be applied directly; however techniques similar to Sharkovsky's were used in 1977 to prove the following surprising relation between Newton's and the Secant iterations:

> Suppose that Newton's iteration, when used to find a zero of f , converges from every starting point in some interval. Then so does the Secant iteration converge from every pair of starting points in that interval, except possibly when f does not reverse sign at its zero. (The exceptional cases are like $f(x) = x^2$ with starting iterates $x_0 = -x_1 = 1$.)

This relation should reinforce a predisposition to use Secant iteration in preference to Newton's except when the derivative f' can be calculated conveniently as well as quickly. For your convenience, Secant iteration programs for several calculators are supplied below. They all work the same way. They require a subroutine that replaces a displayed value x by $f(x)$ to be programmed in place of the ellipsis "...". This subroutine can be invoked independently by pressing

[GSB] 24 on the hp-33 and hp-10C ,
[A] on many other hp calculators older than the hp-28C ,
[A'] on the TI-58 and 59 , so their MA 08 will work too.

To run the iteration, choose initial iterates x_0 and x_1 and key in

x_0 [ENTER] x_1 [R/S] on the hp-33 and hp-10C ,
x_0 [ENTER] x_1 [B] on other hp calculators,
x_0 [x≷t] x_1 [B'] on the TI-58 and 59 .

During the iteration the calculator will pause briefly and display alternately each iterate x to several sig. dec., then its $f(x)$ to three sig. dec. Press [R/S] to stop the iteration when $f(x)$ is negligible or when successive iterates x differ negligibly; and then find the latest iterate in register #1 on hp machines, in #11 on TI machines. The iteration programs use

Registers #1 to 4 on the hp-33 and hp-10C ,
Registers #1 to 3, and Label [O] on other hp calculators,
Registers #11 to 13, and Label [Pause] on the TI-58 and 59 .

(On the TI-58 and 59 registers #0 to 9 and labels [A] to [E] and keys [CLR] and [=] must be left unused if you wish to use the library's zero-finding program MA 08 .)

hp-33/10C		Other hp	TI-58 and 59			
01	STO 2	LBL B	00	LBL	25	Exc
02	CLX	STO 2	01	B'	26	13
03	STO 3	CLX	02	STO	27	x
04	−	STO 3	03	12	28	(
05	PAUSE	LBL 0	04	0	29	RCL
06	STO 1	−	05	STO	30	11
07	GSB 24	PSE	06	13	31	−
08	STO− 3	STO 1	07	x≷t	32	Exc
09	STO 4	GSB A	08	LBL	33	12
10	SCI 2	STO− 3	09	Pause	34	)
11	PAUSE	DSP 2 or SCI 2	10	Pause	35	+
12	SCI 6	PSE	11	STO	36	RCL
13	RCL 1	DSP 9 or SCI 6	12	11	37	11
14	RCL 2	RCL 1	13	A'	38	)
15	RCL 1	RCL 2	14	INV	39	GTO
16	STO 2	RCL 1	15	SUM	40	Pause
17	−	STO 2	16	13	41	LBL
18	RCL 3	−	17	Fix	42	A'
19	÷	RCL 3	18	2	...	...
20	RCL 4	÷	19	Pause	...	...
21	STO 3	R↑	20	Fix	...	INV SBR
22	x	STO 3	21	7		
23	GTO 04	x	22	(		
24	...	GTO 0	23	CE		
...	...	LBL A	24	÷		
...	RTN	:::				
		RTN				

Example: find all real zeros of $f(x) = \exp(x-2) - (x/2)^2$.

Programs:
hp: LBL A ENTER ENTER 2 − exp X≷Y .5 x X² − RTN
TI: LBL A' ((STO 10 − 2) INV lnX − (.5 x RCL 10) X²) INV SBR

Iterations:

N	:	0	1	2	3	4	5	...
x_N	:	0	−1	−0.403	−0.522	−0.5605	−0.556855	...
$f(x_N)$	:	0.135	−0.200	.0497	.0122	−.00126	.0000264	...

N	:	0	1	2	3	4	5	6	...
x_N	:	0	+1	7.753	0.997	0.995	1.8958	1.9155	...
$f(x_N)$	:	0.135	0.118	300.	0.118	0.119	.00253	.00169	...

The BASIC program below performs similarly on an IBM PC , and can be adapted as indicated to do likewise on an HP-71B :
Press [0] [EndLine] x_0 [EndLine] to enter x_0 .
Press [1] [EndLine] x_1 [EndLine] to enter x_1 .
Press [i] [Endline] to Iterate; $(x_{N-1}, x_N) \rightarrow (x_N, x_{N+1})$.
Press [EndLine] to quit.

```
10 ' SECANT ITERATION demonstration program.
20 ' Use DISP, ! , @ instead of PRINT, ' , : resp. on the hp-71B .
30 DEF FNF(X) = EXP(X-2) - (X/2)^2 : ' ... or some other expression in X
40 PRINT "To enter initial iterates x0 or x1 , or to Iterate, ..."
50 FOR I= 1 TO 100 : NEXT I : ' ... pauses.
60  PRINT " ... press [0], [1] or [i] :"; : INPUT K$
70  IF K$="0" THEN GOTO 110
80  IF K$="1" THEN GOTO 170
90  IF K$="i" OR K$="I"  THEN GOTO 150
100 IF K$="" THEN STOP ELSE GOTO 200
110   INPUT "X0 = "; X
120     X0=X : F0 = FNF(X0) : F=F0
130     GOTO 190
140 '
150   X = X1 + (X1-X0)*(F1/(F0-F1))
160     X0=X1 : F0=F1 : GOTO 180
170   INPUT "X1 = "; X
180   X1=X : F1 = FNF(X1) : F=F1
190   DISP  "f(";X;") = "; @ SCI 2 @ DISP F; @ STD !       on hp-71B
190   PRINT "f(";X;") = "; : PRINT USING "##.##^^^^";F; '  on IBM PC
200     GOTO 60 : END : '  Use  50 instead of 60  on  hp-71B .
```

Use a program similar to one of those above to solve all the equations previously presented as examples, namely $x - e^{-x} = 0$, $(x-y) + \ln(y) = 0$ and its transformed form, $3x^5 - 10x^3 + 23x = 0$, $\arctan(x-\pi) = 0$, $x/(1-x) = 0$, $x^{10} = 0$ and $\exp(-x^{-m}) = 0$; try various initial iterates x_0 and x_1 , including when possible some from which Newton's iteration would not converge. (Plot graphs!)

How fast is Secant iteration?

Normally the Secant iteration converges *superlinearly* with *order* $(1+\sqrt{5})/2 = 1.618$. To see what this means, obtain explicit formulas for the Secant iterates x_N in terms of x_0 and x_1 when applied to a typical equation $x/(1-x) = 0$. Those formulas will involve the *Fibonacci* numbers

$F_1 = F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, ...

$$F_N = F_{N-1} + F_{N-2} = \left(((1+\sqrt{5})/2)^N - (-2/(1+\sqrt{5}))^N \right)/\sqrt{5} .$$

Compared with Newton's iteration, which normally converges with order 2 , the Secant iteration takes more iterations to achieve comparable accuracy, but each Secant iteration costs less time. How small must the ratio

(time to compute both f and f') / (time to compute f alone)

be to make Secant iteration ultimately slower than Newton's ?

Both Newton's and the Secant iterations can converge slowly (*linearly*) to a multiple zero of f , as we have seen; but even so the successive values $f(x_N) \to 0$ normally faster than some constant multiple of 2^{-N} as $N \to \infty$. (This was proved just recently.) Consequently, schemes for accelerating convergence to a multiple zero of unknown multiplicity cannot shorten computing times enormously unless the computer has an extremely tiny underflow threshold.

If we are to start from scratch to verify the foregoing claims concerning the speeds at which the Secant and Newton's iterations normally converge, we must first build certain machinery. Ideas due essentially to Hermite will be followed here, though they may seem at first to be a digression.

Let a, b, c, ... be vectors, and let g(x) be a vector-valued function of the vector argument x . Now certain averages of g(x) shall be defined:

$$\begin{aligned} \mathrm{AV}\, g(\langle a,b\rangle) &:= \int_0^1 g(a+\theta(b-a))\, d\theta \\ &= \text{the uniformly weighted average of } g(x) \text{ on the line segment joining } a \text{ to } b \\ &= \mathrm{AV}\, g(\langle b,a\rangle) \ . \end{aligned}$$

$$\begin{aligned} \mathrm{AV}\, g(\langle a,b,c\rangle) &:= 2\int_0^1 \int_0^{\phi} g(a+\theta(b-a)+\phi(c-b))\, d\phi\, d\theta \\ &= \text{the uniformly weighted average of } g(x) \text{ on the triangle whose vertices are } a, b, c \ . \\ &= \mathrm{AV}\, g(\text{ any permutation of } \langle a,b,c\rangle) \ . \end{aligned}$$

Further averages over tetrahedra and other higher-dimensional simplices could be defined this way, but we shall not need them now. Instead a couple of derivatives will be averaged to define

$$\Delta g(\langle a,b\rangle) := \mathrm{AV}\, g'(\langle a,b\rangle) \quad \text{and} \quad \Delta^2 g(\langle a,b,c\rangle) := \mathrm{AV}\, g''(\langle a,b,c\rangle)/2 \ ,$$

called respectively *first* and *second divided differences* of g . The terminology is justified by the observations that

$$\Delta g(\langle a,b\rangle)\,(b-a) = \int_0^1 g'(a+\theta(b-a))\, d\theta\,(b-a) = g(b)-g(a)$$

and similarly

$$\begin{aligned} \Delta^2 g(\langle a,b,c\rangle)\,(c-b)\,(c-a) &= \left(\Delta g(\langle a,c\rangle) - \Delta g(\langle a,b\rangle)\right)(c-a) \\ &= g(c) - g(a) - \Delta g(\langle a,b\rangle)\,(c-a) \ , \end{aligned}$$

whence follows

Newton's Divided-Difference Interpolation Formula:

$$g(x) = g(a) + \Delta g(\langle a,b\rangle)\,(x-a) + \Delta^2 g(\langle a,b,x\rangle)\,(x-b)\,(x-a) \ .$$

When b = a , Newton's formula reduces to Taylor's formula

$$g(x) = g(a) + g'(a)\,(x-a) + \Delta^2 g(\langle a,a,x\rangle)\,(x-a)^2$$

in which the third term, the *remainder* term, is here expressed in terms of a weighted average $\Delta^2 g$ of half the second derivative g" instead of a sample of it somewhere between a and x . Both are *Interpolation* formulas in the sense that the first two terms

$$Pg(x, \langle a,b\rangle) := g(a) + \Delta g(\langle a,b\rangle)\,(x-a) = g(b) + \Delta g(\langle a,b\rangle)\,(x-b)$$

provide a polynomial of lowest degree in x that interpolates (matches) g(x) at x = a and at x = b when b ≠ a , or matches g(x) and g'(x) at x = a when b = a . Pg is easiest to interpret when a, b, x are all scalars, as we shall assume henceforth, in which case the graph of Pg plotted against x is tangent to the graph of g if b = a , secant otherwise.

Taylor's formula will be used to explain how fast Newton's iteration $x_{N+1} := x_N - f(x_N)/f'(x_N)$ normally converges; and then Newton's formula will be used to explain how fast the Secant iteration $x_{N+1} := x_N - f(x_N)/\Delta f(\langle x_N, x_{N-1}\rangle)$ normally converges. Both iterations will be assumed to start from inside some interval $z-\delta \le x \le z+\delta$ surrounding the desired zero z of f and so

narrow that it satisfies the following

Hypothesis: $(\max_Y |f''(y)|)/(\min_X |f'(x)|) < 2/\delta$, where the maximum and minimum are taken over y and x in that interval.

The first conclusion from the hypothesis is that Newton's iteration converges at least *Quadratically* to z from every starting point x_o inside that interval;

$\log(|x_N-z|/\delta) < 2^N \log(|x_o-z|/\delta) \to -\infty$ as $N \to \infty$.

Proof: Taylor's formula says that

$0 = f(z) = f(x_N) + f'(x_N)(z-x_N) + \Delta^2 f((x_N,x_N,z))(z-x_N)^2$,

so $x_{N+1}-z = (x_N-z)^2 \Delta^2 f((x_N,x_N,z))/f'(x_N)$, in which $\Delta^2 f$ is an average of $f''/2$ between x_N and z . Provided x_N lies inside the interval in question, $|\Delta^2 f| \le \max_Y |f''/2| < \min_X |f'|/\delta$, so $|x_{N+1}-z|/\delta < (|x_N-z|/\delta)^2$ and therefore x_{N+1} lies deeper inside the interval. Since x_o was assumed to lie inside the interval, induction proves the same for all subsequent iterates; and a second induction using the logarithm of the last inequality confirms the first conclusion fully.

The second conclusion is that the Secant iteration converges to z from every starting points x_o and x_1 inside that interval, and the order of convergence is at least $(1+\sqrt{5})/2 = 1.618$;

$\log(|x_N-z|/\delta) < F_N \log(|x_1-z|/\delta) + F_{N-1} \log(|x_o-z|/\delta)$ if $N > 1$

$\to -\infty$ faster than some multiple of -1.618^N .

(Here F_N is the N^{th} Fibonacci number introduced earlier.)

Proof: $f(z) = 0$ and Newton's formula says that

$f(z) = f(x_N) + \Delta f((x_N,x_{N-1}))(z-x_N) + \Delta^2 f((x_N,x_{N-1},z))(z-x_N)(z-x_{N-1})$,

so $x_{N+1-z} = (x_N-z)(x_{N-1}-z)\Delta^2 f((x_N,x_{N-1},z))/\Delta f((x_N,x_{N-1}))$. Now Δf is an average of f' inside an interval in which it cannot change sign, because $\min_X |f'| > 0$, so $|\Delta f| \ge \min_X |f'|$. And $|\Delta^2 f| < \min_X |f'|/\delta$ as before. Applying induction to the inference $|x_{N+1}-z|/\delta < |x_N-z|\,|x_{N-1}-z|/\delta^2$ implies that all subsequent iterates x_N lie deeper inside the interval than do x_o and x_1 ; and further inductions applied to the logarithm of the last inequality finish the proof.

A third conclusion to be drawn from the hypothesis above is that the ratios $(x_{N+1}-z)/(x_N-z)^2$ in Newton's iteration, and $(x_{N+1}-z)/((x_N-z)(x_{N-1}-z))$ in the Secant iteration, both approach the same limit $f''(z)/(2f'(z))$ as $N \to \infty$ provided f'' is continuous at z , as is normally the case. This coincidence permits us to compare how fast the iterations converge from the same starting point(s) very close to z , and implies that the Secant iteration takes about 44% more iterations than Newton's to achieve a comparable substantial improvement, say 10 more sig. dec., in accuracy. ($\log(2)/\log(1.618) = 1.44$.)

Practical Considerations:

Other iterative methods exist besides the Secant and Newton's iterations. For example, instead of drawing a secant through two points of the graph of $f(x)$, as the Secant iteration does, a parabola could be drawn through three points. This is what Muller's method does, and it rewards that effort with a higher order of convergence, about 1.8 instead of 1.6 . In general, the more information about $f(x)$ that can be incorporated into an

iterative formula, the faster it should converge, though a law of diminishing returns will ultimately prevail.

An iteration's order of convergence is not all that matters; it affects only the intermediate stage of an attempt to solve an equation. During the early stage, when the iteration is not yet so close to a root that the foregoing theory is applicable, much time can be wasted meandering about aimlessly unless something is done to prevent that. During the later stage, when the iteration is so close to a zero of $f(x)$ that its tiny computed values are obscured by roundoff, much time can be wasted dithering with the last few digits of x unless something is done to prevent that. Rapid convergence during the intermediate stage implies that it cannot last long unless extraordinarily high accuracy is sought.

Most zero-finding software includes some kind of *bracketing* to discourage meandering during the early stage. For instance, if $f(x)$ takes opposite signs at any two iterates, then the software will somehow force all subsequent iterates to lie strictly between those two. Bracketing also discourages dithering in the later stage, but does not prevent it, and can slow convergence.

To inhibit dithering caused by roundoff, most software demands a stopping criterion from its user. That criterion should be composed of two parts; first, a tolerance level for deciding when consecutive iterates x_N are too close together to be regarded as distinguishable; second, a threshold beneath which $|f(x_N)|$ is regarded as negligible. Both the tolerance and the threshold depend upon the level of roundoff. The tolerance must not be smaller than the difference between adjacent representable numbers the computer has available for x, nor need reasonable tolerances much exceed that. The threshold must at least exceed whatever roundoff will contribute to the computed value of $f(x)$, for which an estimate much better than an informed guess may be hard to find.

Because equation-solvers have to stop after only finitely many iterations, they cannot be infallible. Many kinds of failure are possible, but some of them cannot fairly be blamed upon the root-finding software. For example, a root-finder could claim that

$$10^{30}(2(x-0.5) + x) + (10^{-30})/(2(x-0.5) + x)$$

has a zero very near $x = 1/3$, though in fact it has a pole there instead, simply because that expression takes the same values as $10^{30}(2(x-0.5) + x)$ on most computers (unless they carry more than about 20 significant decimals). A root-finder could similarly claim that

$$10^{30}(2(x-0.5) + x)^2 - 10^{-30}$$

has no zero anywhere because it would never change sign unless at least about 30 sig. dec. were carried. These failures arise because the arguments x available to the computer are too sparsely distributed to reveal the intent concealed in those arithmetic expressions. Other failures can occur because the computed values of $f(x)$ are so contaminated by roundoff that they vanish far from a zero of f, or fail to vanish at all. For example (admittedly another very artificial example),

$$(10^{30} + x) - (10^{30} + 1)$$ (Do not omit parentheses!)

is an expression that roundoff forces to vanish for many values of x far from its true zero $x = 1$, unless the computer carries rather more than 30 sig. dec. Similarly, roundoff prevents

$$((10^{19} + x/2)^2 - (10^{19})^2) - 1$$

from ever vanishing; instead, its computed value is -1 over a wide range of positive and negative values of x, and it reverses sign only far from its smaller true zero near 10^{-19}.

Therefore, the findings of a general-purpose root-finder must be treated as food for thought, to be interpreted in the light of information about an equation $f(x) = 0$ that cannot easily be fed to the software. In rare cases, particularly when $f(x)$ has a simple expression whose analytic and computational properties are well understood, better results will be obtained from a simple program implementing the Secant or Newton's iteration than from general-purpose software. More often, whatever is done to transform the equation $f(x) = 0$, and to find starting iterates from which those two iterations would converge quickly to the desired root, will also help general-purpose root-finders work quickly and reliably.

Specialized root-finders exist for special kinds of equations. For example, polynomial equations $\sum_0^N a_j z^{N-j} = 0$ with given numerical values for the coefficients a_j should be solved by specialized software that finds all N zeros z faster than any iteration described above can. The best schemes I know are based upon *Laguerre*'s iteration. Polynomial equations of the form $\det(zI - B) = 0$ with a given square matrix B are *eigenproblems* (a root z is called an *eigenvalue* of B) for which again there exist faster and more accurate methods than any based upon expansion of the determinant as an explicit polynomial, unless B is very *sparse* (elements almost all zeros). The best methods may be found in the *LAPACK* library distributed by the Society for Industrial and Applied Mathematics, among others.

Also special are the equations that *Optimization* calculations must solve to maximize some *objective* function subject usually to constraints upon the unknowns. Once again special methods work best because they need not search for a solution anywhere where an objective function would get worse. Of course, all those special methods may well incorporate ideas discussed in these notes, but details lie far beyond the scope of a first course.

Acknowledgements:
These notes, prepared for an introductory class in Numerical Analysis, include mention of unpublished results discovered by the author in 1977 while enjoying the support of, among others, the U. S. Office of Naval Research, contract N00014-76-C-0013.