The Quadratic Equation Made Difficult

Remember the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

This formula gives the solution of the quadratic equation $ax^2 + bx + c = 0$ in terms of the five operations of addition, subtraction, multiplication, division and square root extraction. Numbers which can be written in terms of these basic five operations are called constructible. For such numbers, it is common to rationalize the denominator so that the only surds which appear are in the numerator. This is the form used for the quadratic formula.

A naive Matlab program to solve quadratic equations might look like

Matlab Example 5a

```
function [x1,x2]=qform5a(a,b,c)
    x1=(-b+sqrt(b*b-4*a*c))/(2*a);
    x2=(-b-sqrt(b*b-4*a*c))/(2*a);
```

After storing the above lines in the file `qform5a`, we may run the above program as

Running Example 5a

```
>> format long
>> [x1,x2]=qform5a(1,100,1)
x1 = -0.0100010002000488
x2 = -99.9899989998000
```

However, solutions to the quadratic equation computed by this program are not as precise as they should be. If b is large compared to a and c, then $\sqrt{b^2 - 4ac} \approx b$. In this case a loss of precision will result from the subtraction of two nearly equal numbers in the numerator of the quadratic formula.

For example, let us solve the quadratic equation

$$x^2 + 100x + 1 = 0$$

using a decimal floating point representation with 3 significant digits. Since

$$\texttt{sqrt}(100^2 -^* 4) = \texttt{sqrt}(1.00 \times 10^4) = 100,$$

then

$$x_1^* = \left(\frac{-100 +^* 100}{2}\right)^* = 0.$$

All significant digits in this approximation x_1^* of x_1 have been lost due to cancellation.

A better approximation can be obtained by rewriting the quadratic formula in a form that avoids the cancellation.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \cdot \frac{-b \mp \sqrt{b^2 - 4ac}}{-b \mp \sqrt{b^2 - 4a}} = \frac{2c}{-b \mp \sqrt{b^2 - 4a}}$$

Using this mathematically equivalent form we obtain

$$x_1^* = \left(\frac{2}{-100 -^* 100}\right)^* = -0.100$$

This approximation is correct to three significant digits, the best we could hope for.

Similar arguments show that the approximation of x_2 is best performed using the original form of the quadratic formula. In fact, if we try to calculate x_2 using the modified version of the quadratic formula that was so successful for x_1, the calculation fails with a division by 0. A correct program for solving the quadratic equation must first test whether b is positive or negative and then use the appropriate form of the quadratic formula that avoids subtraction and the resulting loss of precision for each solution. A Matlab program that employs such a strategy is

Matlab Example 5b

```
function [x1,x2]=qform5b(a,b,c)
    if b>0
        x1=(2*c)/(-b-sqrt(b*b-4*a*c));
        x2=(-b-sqrt(b*b-4*a*c))/(2*a);
    else
        x1=(-b+sqrt(b*b-4*a*c))/(2*a);
        x2=(2*c)/(-b+sqrt(b*b-4*a*c));
    end
```

More complicated formulae for solving cubic and quartic equations exist. The appropriate algebraic transformation to avoid loss of precision when using these formulae for numerical calculation are also more complicated. Moreover, it was proved by Abel in 1824 that no general formula exists for the solving polynomial equations of degree 5 or higher. Therefore more general methods are needed for solving these and other equations.

The intermediate value theorem guarantees that if $f : [a, b] \to \mathbf{R}$ is a continuous function such that $f(a)$ and $f(b)$ have opposite signs, then there exists $x \in (a, b)$ such that $f(x) = 0$. This observation leads to a simple method of searching for roots of a continuous function called the bisection method or binary search.

The bisection method proceeds as follows: Guess the root of f to be the midpoint $c = (a + b)/2$ of the interval $[a, b]$. If $f(c) = 0$, then we're lucky and there is no need to continue. If $f(c)$ has the opposite sign of $f(a)$ then there must be a root in $[a, c]$ so we continue by looking for a root there. If $f(c)$ has the opposite sign of $f(b)$ then we continue looking for the root in $[c, b]$.

At each step the length of the interval under consideration is cut in half. Moreover, the intermediate value theorem guarantees that there is a solution to $f(x) = 0$ in that interval. In particular, after n iterations we know that the solution x is contained in an interval of length $(b - a)/2^n$. Therefore, by approximating x by the midpoint $x^* = c$ of the n-th interval, we obtain that $|x^* - x| < (b - a)/2^{n+1}$.

How many iterations to make is dependent on the accuracy of the solution required.

A Matlab program that finds an approximate solution x^* with absolute error less than ϵ is

Matlab Example 5c

```
function x=bisect5c(f,a,b,epsilon)
    e2=epsilon*2;
    fa=feval(f,a);
    fb=feval(f,b);
    if fa*fb>=0 || b<=a
        disp('Not a good initial bracket!');
    end
    while b-a >= e2
        c=(a+b)/2;
        fc=feval(f,c);
        if fc==0.0
            x=c;
            return
        end
        if fc*fa>0
            a=c;
        else
            b=c;
        end
    end
    x=(a+b)/2;
```

After storing the above lines in the file `bisect5c`, we may run the above program as

Running Example 5c

```
>> bisect5c('sin',1,4,0.001)
ans = 3.14141845703125
>> f=inline('x*x+100*x+1');
>> bisect5c(f,-1,0,0.001)
ans = -0.0100097656250000
```

Note, that since the interval $[a, b]$ itself must be represented using a floating point approximation $[a^*, b^*]$, then there is a point after which it can't be cut in half again. In particular, after a certain number of iterations the approximate bisecting operation

$$c^* = \left(\frac{a +^* b}{2}\right)^*$$

will be rounded so that c^* is equal to either a^* or b^*. At this point the bisection algorithm must stop—we have reached machine precision.

We seldom need machine precision. However, if machine precision is required, the

Matlab code can be modified to use $c^* = a^*$ or $c^* = b^*$ as a stopping condition. We obtain

Matlab Example 5d

```
function x=bisect5d(f,a,b)
    fa=feval(f,a);
    fb=feval(f,b);
    if fa*fb>=0 || b<=a
        disp('Not a good initial bracket!');
    end
    c=(a+b)/2;
    while c ~= a && c ~= b
        fc=feval(f,c);
        if fc==0.0
            x=c;
            return
        end
        if fc*fa>0
            a=c;
        else
            b=c;
        end
        c=(a+b)/2;
    end
    x=(a+b)/2;
```

This program approximates the solution x to $f(x) = 0$ as accurately as possible. For example, we may approximate π by solving $\sin(x) = 0$ on the interval $[1, 4]$ as

Running Example 5d

```
>> bisect5d('sin',1,4)
ans = 3.14159265358979
>> pi
pi = 3.14159265358979
>> bisect5d(f,-1,0)
ans = -0.0100010002000500
```

As a final note, depending on the programming language, complier, and compiler optimizations used, a stopping condition that depends on an exact equality like the one used in Example 5d can sometimes fail to work properly. Suppose a and b are stored as double precision floating point numbers and that the floating point registers of the FPU work in extended double precision. This is the default for many Intel FPUs. If $c = (a+b)/2$ is computed using FPU registers in extended double precision, then it will never be equal to the double precision values of a and b. In this case the above program might enter an infinite loop. Similarly, testing for equality between a single precision and a double precision variable is almost always sure to fail. As a general rule it is usually best to avoid testing for exact equality between any two floating point numbers.