Department of Computer Science,
University of Otago

Te Whare Wānanga o Otāgo

Technical Report OUCS-2002-02

Integer root finding, a functional perspective

Author:

Michael H. Albert

Status: submitted to the *Journal of Functional Programming* for a special issue on "functional pearls"

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/trseries/

FUNCTIONAL PEARLS

Finding integer roots, a functional perspective.

MICHAEL H. ALBERT

Dept. of Computer Science, University of Otago, Dunedin, New Zealand
(*e-mail:* malbert@cs.otago.ac.nz)

Abstract

We consider the problem of finding integer parts of roots to equations in a functional context. Standard techniques from numerical methods are extended to this case, providing both good performance, and guarantees of correctness. From an educational standpoint, the clear connection between the methods for finding roots and the functional code makes it easier to understand how different methods work, and to compare their relative effectiveness.

1 Introduction

This article had its genesis in the following question set as an exercise in a functional programming course:

A triangle with sides a, b, and c has area A, given by Heron's formula:

$$\begin{aligned} s &= \frac{a+b+c}{2} \\ A &= \sqrt{s(s-a)(s-b)(s-c)} \\ &= \frac{\sqrt{(a+b+c)(a+b-c)(b+c-a)(c+a-b)}}{4}. \end{aligned}$$

Given an integer n find all triangles whose sides all have integer length at most n and whose area is also an integer.

Aside from some, deliberate, ambiguity as to whether $(3,4,5)$ and $(4,5,3)$ count as different triangles (they don't), the exercise was intended to be a basic test of understanding list comprehension. A possible solution (in Haskell) is given in Figure 1. The only difference between it and the actual specification of the relations above is the observation that in order for A to be an integer it is both necessary and sufficient that $a+b+c$ is even, and the numerator of the final expression is an integer. The slight type awkwardness in `iSqrt` is foreshadowed in section 12.2 of the *Gentle Introduction to Haskell* (Hudak, Peterson and Fasel, 1987).

Checking whether a number is a perfect square by taking a floating point square root, truncating, squaring, and checking against the original is of course a standard trick but it limits the applicability of this method. For example, `isSquare` will report that 2^{101} is a perfect square. On the other hand, the problem of finding an

```
top a b c    = (a + b + c)*(a + b - c)*(b + c - a)*(c + a - b)
isSquare x   = x == (iSqrt x)^2
iSqrt x      = truncate (sqrt (fromIntegral x)) :: Int

tris n       = [ (a, b, c) | a <- [1..n],
                             b <- [a..n],
                             c <- [b.. (min ((a+b)-1) n)],
                             (a + b + c) `mod` 2 == 0,
                             isSquare (top a b c)
               ]
```

Fig. 1. Triangles with integer areas.

integer square root, using only integer arithmetic has a long history and was used as an example in Dijkstra (1976). A hint of the methods used in that algorithm (which was tailored to fast binary arithmetic) will remain in our methods for producing an initial approximation when searching for the integer part of a solution to more general equations.

General problem: *Given a function f on the real numbers, which is integer-valued on the integers, and a (large) integer b, find the integer part of a solution to $f(x) = b$.*

We will develop a number of methods for attacking this general problem in a context where we can guarantee that there is a unique solution, as of course in complete generality it is insoluble. A not inconsiderable side effect of this development will be to show how easy it is both to code, and to understand the code for, basic numerical methods in the functional context. This should be contrasted with the situation in imperative languages, as seen for example in *Numerical Recipes in C* (Press, Flannery, Teukolsky, Vetterling 1993), which will be our standard source throughout for numerical methods, specifically Chapter 9 which deals with root finding. The novelty of our discussion lies in the fact that *all* the computations are restricted to dealing with integer values. In fact, except for the computation of f itself, we make use only of basic arithmetic operations.

2 Generalities

Most root finding methods operate according to the following general scheme:

- Produce an initial approximation, or approximations, to a root (often a pair of values bracketing a root).
- Update the approximation according to a specified method,
- until some termination check (generally a tolerance, assumed or guaranteed) is satisfied.
- Report the final result.

It is clear that each of these steps corresponds to a natural functional operation, the combination of the second and the third being captured in the higher order `until`

function. The entire scheme can be captured in a "one-liner" assuming suitable definitions for the individual parts:

```
findRoot f = report (until termCheck (iterMethod f) (initApprox f))
```

The importance of the initial step is often overlooked. Many effective root finding algorithms consist of a method which produces a reasonable initial approximation (generally with linear convergence), followed by a few steps of some superlinear method to polish the solution. Recall that, an approximation method is *linear* if it adds a constant number of bits of accuracy at each iteration. Equivalently, the error guarantee in the approximation is divided by a constant factor. More generally, a method has *order of convergence*, c, if each step reduces the error from ϵ to (a constant times) ϵ^c. For *superlinear* methods, $c > 1$.

In order to keep our discussion concrete, and to allow it to fit into the scope of this article, we are now going to make a major technical assumption about the form of the function f in the equations which we are trying to solve. Namely, in looking for (the integer part of) a solution to $f(x) = b$ we require that: the equation has a unique positive solution α, and for some $\beta < [\alpha]$, f is increasing and convex on $[\beta, \infty]$. Our aim is to determine $[\alpha]$, the integer part of α. Although we would like to emphasise the role played by b in ensuring that the conditions are met, it will simplify the code if we express the methods as "root-finders", where the function for which we are seeking a root is $f(x) - b$. Of course, given a `findRoot` function, it is trivial to define a corresponding `solve` function, so this is really no restriction in generality.

The assumptions we make on the form of f may seem to be, and indeed are, quite restrictive. However, they also fit well into the context where methods of this sort would be appropriate as opposed to making use of standard numerical methods packages. Specifically, while the coefficients of f (were it say, a polynomial) would ordinarily be machine size integers, we have in mind that b (and hence α) might well be much larger than that. A typical problem, which we shall use as an example later, would be to find the integer part of the solution to:

$$x^3 = 2^{2003}.$$

In such an instance the assumption concerning the growth rate of f near α is easily justified. In particular, the problem of finding the integer part of the nth root of a large integer, or for that matter any positive integer, satisfies the assumptions, and the reader may, if he or she wishes, consider the subsequent discussion as applicable only to that case. In our general scheme of root-finding methods, both the termination check and the initial approximation may depend on the iterative method used. For floating point computations, the termination check generally reduces to seeing that the approximations to a root have "settled down". As we will be demanding exact computation of the integer part of the root, ours will in some sense be more stringent. In fact the termination checks are quite closely bound with the iterative method itself, while the initial approximation generally depends on it in a more technical fashion. So we will begin with a discussion of the iterators

```
stepB f (low, high)
    | (f mid) <= 0   = (mid, high)
    | otherwise     =  (low, mid)
    where mid = (low + high) `div` 2

findRootB f (l0,h0)  = fst (until termCheckB(stepB f) (l0,h0))
     where termCheckB (l,h) = (h - l) <= 1
```

Fig. 2. Bisection method with a fixed initial interval.

and their termination, and then move to a method for generating suitable initial approximations.

3 Iterators and termination

We will confine our detailed consideration of iteration methods to bisection, Newton's method and the secant method. Any of the other methods considered in Chapter 9 of (Press, Flannery, Teukolsky, Vetterling 1993) could be used as well, and some would be more appropriate in other contexts, but the basic ideas are the same. Some care must be used in choosing a method, as it should be remembered that in computations with large integers, the cost of various operations may be quite different from one another.

Bisection is the most fundamental of the root finding methods, and one of the only ones which, for continuous functions in general, is guaranteed to succeed. Our main use for bisection will be in constructing decent initial approximations. The idea of bisection is of course very simple. At each stage we have an interval $[l, h]$ with $f(l) \leq 0 < f(h)$. We insist on the strict upper inequality because we know that there is a unique root in the interval we are searching, and we wish to find its integer part. We then evaluate f at the midpoint of the interval, and construct a new interval with the same bracketing property depending on the value of f at the midpoint. Generally, the termination condition is taken to be that the length of the interval $[l, h]$ is suitably small. If we seek to find the integer part of a solution, that amounts to insisting that $h - l \leq 1$. The complete code for bisection (assuming an initial interval $[l_0, h_0]$) is given in Figure 2.

In the real-valued setting, Newton's method constructs a sequence of approximations x_n to a solution of $f(x) = b$. After choosing some initial approximation (read, guess) x_0, each new approximation x_{n+1} is taken to be the x-intercept of the intersection of the tangent line to f through $(x_n, f(x_n))$. That is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We have assumed that f is convex to the right of α. Therefore, if $x_n > \alpha$ it will also be the case that $x_{n+1} > \alpha$. Since we wish to carry out all our computations in integer arithmetic we must assume that f' is also integer-valued on the integers,

```
stepN f fp x     = x - (f x `cdiv` (fp x))
iterN f fp (x,_) = (stepN f fp x, x)
termN (x,y)      = (x >= y)

n `cdiv` q = 1 + (n-1) `div` q
```

Fig. 3. Newton iteration and termination.

and we set:

$$x_{n+1} = x_n - \left\lceil \frac{f(x_n)}{f'(x_n)} \right\rceil .$$

It follows that if $\alpha < x_n$ then $[\alpha] \leq x_{n+1} < x_n$. On the other hand if $x_n = [\alpha]$ then $x_{n+1} \geq x_n$ (since $f(x_n) \leq 0$, while $f'(x_n) > 0$) so our termination condition is simply that $x_{n+1} \geq x_n$. Since the termination condition requires us to compare two approximations, we must build this into our iterator. This leads to the code shown in Figure 3. That the function `cdiv` computes the ceiling we require follows as the dividend is an integer. This code assumes that the function `f` and its derivative `fp` are given. In practice (for example for computing nth roots) it might well be efficient to compute these two values together, modifying the `stepN` method appropriately. When the initial approximation is sufficiently good, Newton's method converges quadratically, that is, it roughly doubles the number of digits of accuracy with each iteration. It can be checked that for polynomial functions of degree n it is sufficient that the initial error must be at most α/n to ensure quadratic convergence. This is easily achieved by at most $\lceil \log_2 n \rceil$ bisection steps, beginning from a bracketing pair $2^{k-1} < \alpha \leq 2^k$. Therefore our goal for finding an initial approximation will be to determine this value of k. In practice, the refinement of further bisection is generally unnecessary, as the quadratic regime is entered after a few steps, even from an initial approximation of 2^k.

Although Newton's method guarantees quadratic approximation, each step requires an evaluation of both the function and its derivative. For polynomials, we can combine the evaluation in such a way that the total cost will not be much greater than a single function evaluation, but for other functions this may not be possible. In that case the secant method may be more appropriate. In this method, a secant is drawn connecting two points on the curve (representing the two most recent approximations) and its x-intercept is taken as the next approximation. Again, since we are assuming that the function is convex and increasing to the right of α this intercept will also lie to the right of α. So, if $\alpha < x_n < x_{n-1}$ then we may take:

$$x_{n+1} = x_n - \left\lceil \frac{f(x_n)(x_{n-1} - x_n)}{f(x_{n-1}) - f(x_n)} \right\rceil .$$

and we can guarantee that $[\alpha] \leq x_{n+1} < x_n$. On the other hand if $x_n = [\alpha]$ then $x_{n+1} \geq x_n$, so our termination condition is again simply that $x_{n+1} \geq x_n$. Since the values of f at both x_n and x_{n-1} are required to compute x_{n+1}, it makes sense to

```
stepS (x1, f1, x0, f0) = x1 - (f1*(x0 - x1) `cdiv` (f0 - f1)) - 1
iterS f old@(x1, f1, x0, f0) = (x2, f2, x1, f1)
         where x2 = stepS old
               f2 = f x2
termS (x,_,y,_) = (x >= y)
```

Fig. 4. Secant iteration and termination.

include them as part of the iterator, and this is done in Figure 4 (the definition of `cdiv` is the same as in Newton's method).

Under our standing conditions, the secant method is often more effective than Newton's method. The theoretical order of convergence is equal to the golden ratio (1.618...), which is less than the quadratic convergence given by Newton's method but, as noted above, if the evaluation of $f'(x)$ cannot be carried out with little additional computation over the evaluation of $f(x)$, then this assessment of the Newton method's convergence is overly generous. In the case where evaluating f and f' are separate operations of equal complexity, the actual order of convergence for Newton's method is $\sqrt{2}$ which is dominated by that of the secant method.

A third method called Ridder's method is discussed in *Numerical Recipes* and would be particularly appropriate for handling functions of exponential growth. However, for such functions, the solution is likely to be small, for example:

What is the largest value of n for which

$$n^2 2^n + 3^n < 10^{1000}?$$

So, for such equations, when searching for integer solutions, more straightforward methods such as bisection are probably preferable. Obviously exceptions could arise if the evaluation of the function were particularly complex.

4 The initial approximation

In all cases, we will want to initially generate an approximation to the root which is larger than the root and has, within one, the correct number of binary digits. Our assumptions ensure that for $0 < x < \alpha$, $f(x) < 0$, so it is a simple matter of checking powers of two until we find the least k with $f(2^k) > 0$. Under most circumstances:

```
initApprox f = head [x | x <- map (2^) [1..], f x > 0]}
```

will be a perfectly satisfactory solution to this problem. We can, however, easily add a little finesse to this method. Instead of constructing the powers of two iteratively, we can square each previous approximation, until we obtain a positive value for f. That is, we have found the least m with

$$f(2^{2^m}) > 0.$$

```
initApprox f = 2^(1 + solveB g (initInterval g (1, 2)))
     where g k = f (2^k)

initInterval  g i@(_, m)
    | g m > 0   = i
    | otherwise = initInterval g (m, 2*m)
```

Fig. 5. Generating an initial approximation.

Now we can proceed with bisection in the exponent on the range $[2^{m-1}, 2^m]$ to zoom in on the proper value of k. This is illustrated in Figure 5. We temporarily introduce a function $g(t) = f(2^t)$ and then successively double the value of t until we overshoot 0. Then `solveB` finds the exact value of k for which $f(2^{k-1}) < 0 \leq f(2^k)$. In very simple examples, such as $f(x) = x^n$, we could make use of extra properties of the function (namely that $f(x^2) = f(x)^2$) to reduce the cost of setting up the initial approximation.

For the secant method this is more or less all we need – we can use the pair 2^k and $2^k - 1$ as our two initial ordinates for beginning the secant method. As noted above, for Newton's method we may, for polynomials of degree n wish to carry out a further $\log_2 n$ steps of the bisection method in order to guarantee subsequent quadratic convergence.

We note, as a matter of purely academic interest, the possibility of bootstrapping our methods for producing an initial approximation. That is, the initial approximation 2^k which we seek for the root of f is just the ceiling of the solution to $f(2^x) = b$, or as noted above, to $g(x) = b$ where $g(x) = f(2^x)$. So, we could produce an initial approximation for the root of g in the same way, then solve for the root of g by any of our chosen methods. Or, we could take this as many levels deeper as we might wish, though we suspect that a search for solutions larger than $2^{2^{2^{16}}}$ is not likely to be practical!

5 Example

It is easy enough to add a counter to any of the methods in order to keep track of the number of function evaluations required. Consider, for the sake of example, the problem of finding the integer part of $2^{667}\sqrt[3]{4}$, that is the integer part of the solution of:

$$x^3 = 2^{2003}.$$

Eleven function evaluations will establish that the solution lies between 2^{512} and 2^{1024}. Ten more will find the upper bound of 2^{668}. Now the three methods diverge. Ordinary bisection is working on an interval of length 2^{667} and will require a further 668 evaluations. Newton's method, beginning directly from this point requires a further nine evaluations (but of both the function and its derivative). The secant method requires 13 steps, but each of these requires only one function evaluation.

For the benefit of the terminally curious, this 204 digit number is:

97206156510086514169078183897808012552541162670116171040103578843383025139468163310215076712832058873615742223501578610875862048824786326698186093069603010997511851065877793612463007752980318968365577 5

If the exponent on the right hand side is changed from 2003 to 20003, then the initial phase requires 27 steps (instead of 21) but Newton's method still requires only a further 12 evaluations after constructing the initial guess. The secant method requires 18 further evaluations after the initial phase, while of course bisection lags in the rear, now requiring 6668 further evaluations. The three extra evaluations required by Newton's method when the number of digits in the solution has been multiplied by 10 is in surprisingly direct accord with the quadratic convergence of this method.

6 Conclusions

Although this presentation is geared towards the one variable setting, the general ideas apply equally well in higher dimensions, given suitable assumptions about the behaviour of the functions involved. Likewise, the methods of chapter 10 in *Numerical Recipes*, for finding maximum or minimum function values can also be attacked in the same fashion.

The discussion above illustrates once again how functional programming techniques provide the glue that allow one to put together different parts of a problem solution. In particular, different iteration methods, termination checks, or generators for initial approximations can be plugged into the method as a whole to illustrate or refine its effectiveness. It would be nice to be able to claim that polymorphism also allows us to use the same methods for floating point solutions as for integer solutions, but as it stands this is not the case owing to our use of `cdiv`. On the other hand, to produce floating point code, only requires changing the declaration of that operation to `cdiv = /`.

We hasten to admit that we have not provided general purpose equation solving routines. Much of the difficulty in providing such methods is in dealing with functions which behave poorly for one method or another. For example, one practical method involves choosing either a secant step, or a bisection step if the secant step is not more effective. Such behaviour could easily be built in through a construct such as `better stepB stepS` requiring only a suitable definition of `better` and a matching of the input type for the two methods.

References

Dijkstra, E. W. (1976) *A Discipline of Programming*. Prentice-Hall.

Hudak, P., Peterson, J. and Fasel, J. (1999) A gentle introduction to Haskell (Version 98), `http://haskell.org/tutorial`.

Press, W. H., Flannery, B. P., Teukolsky, S. A. and Vetterling, W. T. (1993) *Numerical Recipes in C The Art of Scientific Computing (2nd edition)*. Cambridge University Press, Cambridge.