COS 126: Spring 1997

Extracting Roots

Due: Wednesday 9/24

Implement C functions that find can find the roots of some mathematical functions, using the bisection method and Newton's method.

We restrict the class of mathematical functions to be decreasing functions over a given interval, with exactly one root in that interval. That is, given `l`, `r`, and `f(x)`, we will consider only functions for which `x` less than `y` implies that `f(x)` is greater than `f(y)` and for which `f(t) = 0` for exactly one `t` between `l` and `r`. The goal is to find `t`.

This problem is well-studied and has many applications. For example, if we take the interval `[1, 2]` and `f(x) = 4 - x^3`, the root is the cube root of 4. In this way, this program can be used to calculate the `n`th root of any number. For another example, consider the problem of finding the maximum value of a function. This is the same as finding the root of its derivative.

The following program finds the square root of 2, accurate to within two decimal places, using a brute-force method:

```
#include <stdio.h>
float f(float x)
  { return 2.0 - x*x; }
float root(float l, float r)
  {
    float t, epsilon;
    epsilon = .01;
    for (t = l; t <= r; t += epsilon)
      if (f(t) < 0.0) return t;
    return t;
  }
main()
  {
    printf("%f\n", root(1.0, 2.0));
  }
```

This method is not an efficient way to get accurate answers: if `epsilon` is very small, the loop will iterate an enormous number of times. Your task is to get accurate answers (down to `epsilon = .000001`) using a reasonably small number of function evaluations.

The program is organized so that different functions can be tried just by changing the definition of `f`, and perhaps the initial interval endpoints used in the call to `root`.

First, implement the *bisection* method. Check the value of the function at the middle of the interval: if it is positive, replace the left endpoint with the middle point; if it is negative, replace the right endpoint with the middle point. This halves the size of the interval. Stay in a loop doing this until the interval size is less than `epsilon`. Print out all function evaluations to trace the execution of your program.

By editing it to change the function, use your program to find the square root of 2, the 7th root of 126 and the positive real root of `p(x) = 6 - x - x^3`. Debug your program with `epsilon = .01`. Include in your `readme` file the output for a run computing the root of `6 - x - x^3` with `epsilon = .000001`. For the second part of your assignment, package your implementation into a function that computes the square root of a given number. Use this function in a main program that prints out a table of the square roots of the integers from 2 to 20, along with the total number of function evaluations required to compute the table. (The easiest way to get this total is to use a global variable.)

Third, implement *Newton's method* for finding the square root (the case when `f(x) = c - x^2`). This method involves making an initial guess `t`, then computing a new guess by averaging `t` and `c/t`, continuing in this way until two successive guesses are within `epsilon` of one another. Debug the program by printing out all function evaluations, as above, but hand in a run that uses this program to print out a table of the square roots of the integers from 2 to 20, along with the total number of iterations used to compute the table.

Put the three programs that you write in three separate files: `bisect.c` with the bisection method computing the root of `6 - x - x^3`; `bisect2.c` with the bisection method computing the table of square roots; and `newton2.c` with Newton's method computing the table of square roots. Create a `readme` file that has the results of your runs, describes your programs, comments on the results, and describes any problems that you might have had. To electronically submit these files, type `/u/cs126/bin/submit 1 readme bisect.c bisect2.c newton2.c`. You can find more details about submissions on the course home page under Programming Assignments.

You may wish to use `bc` to check your answers.

Extra Credit: First, make your bisection program take a third argument, the function. Then include `math.h` and find the positive roots of some more interesting functions involving trigonometric functions, exponentials, or logarithms. For example, try computing the root of `2 - exp(x)`. (This is a way to compute `ln(2)`.)

Challenge for the bored: Generalize your function for Newton's method to find the `n`th root of any number `c`: the general formula for Newton's method is to replace the guess `t` by `t - f(t)/f'(t)`. (Exercise: check that this reduces to the above method for `f(x) = c - x*x`.) Newton's method is very fast for some functions, but sometimes doesn't converge. In real applications, a combination of these to methods is used: bisection to get near the answer, Newton's method to zoom in on an accurate answer.