

### §3.1 Bisection

We are now concerned with the problem of finding a root for the function  $f(x)$ , *i.e.*, some  $c$  such that  $f(c) = 0$ .

The simplest method is that of bisection. The following theorem, from calculus class, insures the success of the method

**Theorem 1 (Intermediate Value Theorem).** If  $f(x)$  is continuous on  $[a, b]$  then for any  $y$  such that  $y$  is between  $f(a)$  and  $f(b)$  there is a  $c \in [a, b]$  such that  $f(c) = y$ .

The IVT is best illustrated graphically. Note that continuity is really a requirement here—a single point of discontinuity could ruin your whole day, as the following example illustrates.

**Example 2.** The function  $f(x) = \frac{1}{x}$  is not continuous at 0. Thus if  $0 \in [a, b]$ , we *cannot* apply the IVT. In particular, if  $0 \in [a, b]$  it happens to be the case that for *every*  $y$  between  $f(a), f(b)$  there is no  $c \in [a, b]$  such that  $f(c) = y$ .

In particular, the IVT tells us that if  $f(x)$  is continuous and we know  $a, b$  such that  $f(a), f(b)$  have different sign, then there is some root in  $[a, b]$ . A decent estimate of the root is  $c = \frac{a+b}{2}$ . We can check whether  $f(c) = 0$ . If this does not hold then one and only one of the two following options holds:

1.  $f(a), f(c)$  have different signs.
2.  $f(c), f(b)$  have different signs.

We now choose to recursively apply bisection to either  $[a, c]$  or  $[c, b]$ , respectively, depending on which of these two options hold.

**Modifications** Unfortunately, it is impossible for a computer to test whether a given black box function is continuous. Thus malicious or incompetent users could cause a naïvely implemented bisection algorithm to fail. There are a number of easily conceivable problems:

1. The user might give  $f, a, b$  such that  $f(a), f(b)$  have the same sign. In this case the function  $f$  might be legitimately continuous, and might have a root in the interval  $[a, b]$ . If, taking  $c = \frac{a+b}{2}$ ,  $f(a), f(b), f(c)$  all have the same sign, the algorithm would be at an impasse. We should perform a “sanity check” on the input to make sure  $f(a), f(b)$  have different signs.
2. The user might give  $f, a, b$  such that  $f$  is not continuous on  $[a, b]$ , moreover has no root in the interval  $[a, b]$ . For a poorly implemented algorithm, this might lead to an infinite search on smaller and smaller intervals about some discontinuity of  $f$ . In fact, the algorithm might descend to intervals as small as machine precision, in which case the midpoint of the interval will, due to rounding, be the same as one of the endpoints, resulting in an infinite recursion.

3. The user might give  $f$  such that  $f$  has no root  $c$  that is representable in the computer's memory. Recall that we think of computers as storing numbers in the form  $\pm r \times 10^k$ ; given a finite number of bits to represent a number, only a finite number of such numbers can be represented. It may legitimately be the case that none of them is a root to  $f$ . In this case, the behaviour of the algorithm may be like that of the previous case. A well implemented version of bisection should check the length of its input interval, and give up if the length is too small, compared to machine precision.

Another common error occurs in the testing of the signs of  $f(a), f(b)$ . A slick programmer might try to implement this test in the pseudocode:

```
if (f(a)f(b) > 0) then ...
```

Note however, that  $|f(a)|, |f(b)|$  might be very small, and that  $f(a)f(b)$  might be too small to be representable in the computer; this calculation would be rounded to zero, and unpredictable behaviour would ensue. A wiser choice is

```
if (sign(f(a)) * sign(f(b)) > 0) then ...
```

where the function  $\text{sign}(x)$  returns  $-1, 0, 1$  depending on whether  $x$  is negative, zero, or positive, respectively.

We give our algorithm as follows

**Algorithm 1:** Algorithm for finding root by bisection.

**Input:** a function, two endpoints, and a tolerance

**Output:** a point for which the function has small value.

`RUN_BISECTION( $f, a, b, tol$ )`

- (1) Let  $fa \leftarrow f(a), fb \leftarrow f(b)$ .
- (2) **if**  $\text{sign}(fafb) > 0$
- (3)     throw an error.
- (4) **else if**  $\text{sign}(fafb) = 0$
- (5)     **if**  $\text{sign}(fa) = 0$
- (6)         **return**  $a$
- (7)     **else**
- (8)         **return**  $b$
- (9) **else**
- (10)     **return** `recursive_bisection( $f, a, b, fa, fb, tol$ )`

We can see that each time `recursive_bisection( $f, a, b, \dots$ )` is called that  $|b - a|$  is half the length of the interval in the previous call. Formally call the first interval  $[a_0, b_0]$ , and the first midpoint  $c_0$ . Let the second interval be  $[a_1, b_1]$ , etc. Note that one of  $a_1, b_1$  will be  $c_0$ , and the other will be either  $a_0$  or  $b_0$ . We are claiming that

$$\begin{aligned} b_n - a_n &= \frac{b_{n-1} - a_{n-1}}{2} \\ &= \frac{b_0 - a_0}{2^n} \end{aligned}$$

**Algorithm 2:** Recursive part of bisection algorithm.

**Input:** a function, two endpoints, the value of the function at two endpoints, and a tolerance

**Output:** a point for which the function has small value.

RECURSIVE\_BISECTION( $f, a, b, fa, fb, tol$ )

```
(1)   Let  $c \leftarrow \frac{a+b}{2}$ 
(2)   if  $b - a < 2\epsilon$ 
(3)       return  $c$ 
(4)   Let  $fc \leftarrow f(c)$ .
(5)   if  $|fc| < tol$ 
(6)       return  $c$ 
(7)   if  $\text{sign}(fafc) < 0$ 
(8)       return recursive_bisection( $f, a, c, fa, fc, tol$ )
(9)   else
(10)      return recursive_bisection( $f, c, b, fc, fb, tol$ )
```

**Theorem 3 (Bisection Method Theorem).** If  $f(x)$  is a continuous function on  $[a, b]$  such that  $f(a)f(b) < 0$ , then after  $n$  steps, the algorithm `run_bisection` will return  $c$  such that  $|c - c'| \leq \frac{b-a}{2^n}$ , where  $c'$  is some root of  $f$ .