Notes for Lecture 7

Fri, 9/5/2025

Solving equations

Now that we have discussed how computers deal with numbers, it is natural to think about how to compute numbers of interest. Often, these arise as solutions of equations.

For instance. As simple but instructive instances, how do we compute numbers like $\sqrt{2}$, $\log(3)$ or π?

Note that any equation can be put into the form $f(x) = 0$ where $f(x)$ is some function. Solving that equation is equivalent to finding roots of that function.

There are many approaches to root finding see, for instance:

https://en.wikipedia.org/wiki/Root-finding_algorithms

Comment. The solve routines implemented in professional libraries often use hybrid versions of the methods we discuss below (as well as others). For instance, Brent's method (used, for instance, in MATLAB, PARI/GP, R or SciPy) is a hybrid of three: the bisection and secant methods as well as inverse quadratic interpolation.

Comment. It depends very much on $f(x)$ which approach to root finding is best. For instance, is $f(x)$ a nice (i.e. differentiable) function? Is it costly to evaluate $f(x)$? This is the reason for why there are many different approaches to finding roots and why it is important to understand their strenghts and weaknesses.

The bisection method

Suppose that we wish to find a root of the continuous function $f(x)$ in the interval $[a, b]$.

If $f(a) < 0$ and $f(b) > 0$, then the **intermediate value theorem** tells us that there must be an $r \in [a, b]$ such that $f(r) = 0$. (Likewise if $f(a) > 0$ and $f(b) < 0$.)

Comment. Recall that the intermediate value theorem requires $f(x)$ to be continuous so that there are no jumps or singularities.

The **bisection method** now cuts the interval $[a, b]$ into two halves by computing the **midpoint** $c = \frac{a+b}{2}$. Depending on whether $f(c) \leqslant 0$ or $f(c) \geqslant 0$, we conclude that there must be a root in $[a, c]$ or in $[c, a]$. In either case, we have cut the length of the interval of uncertainty in half.

This process is then repeated until we have a sufficiently small interval that is guaranteed to contain a root of $f(x)$.

Example 33. Determine an approximation for $\sqrt[3]{2}$ by applying the bisection method to the function $f(x) = x^3 - 2$ on the interval $[1, 2]$. Perform 4 steps of the bisection method.

Comment. Note that it is obvious that $1 < \sqrt[3]{2} < 2$ so that the interval $[1, 2]$ is a natural choice.

Solution. Note that $f(1) = -1 < 0$ while $f(2) = 6 > 0$. Hence, $f(x)$ must indeed have a root in the interval $[1,2]$.

- $[a,b] = [1,2]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).

  The midpoint is $c = \frac{a+b}{2} = \frac{1+2}{2} = \frac{3}{2}$. We compute $f(c) = c^3 - 2 = \frac{27}{8} - 2 = \frac{11}{8} > 0$.

  Hence, $[a,c] = \left[1, \frac{3}{2}\right]$ must contain a root of $f(x)$.

- $[a,b] = \left[1, \frac{3}{2}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).

  The midpoint is $c = \frac{a+b}{2} = \frac{1+3/2}{2} = \frac{5}{4}$. We compute $f(c) = -\frac{3}{64} < 0$.

  Hence, $[c,b] = \left[\frac{5}{4}, \frac{3}{2}\right]$ must contain a root of $f(x)$.

- $[a,b] = \left[\frac{5}{4}, \frac{3}{2}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).

  The midpoint is $c = \frac{a+b}{2} = \frac{11}{8}$. We compute $f(c) = \frac{307}{512} > 0$.

  Hence, $[a,c] = \left[\frac{5}{4}, \frac{11}{8}\right]$ must contain a root of $f(x)$.

- $[a,b] = \left[\frac{5}{4}, \frac{11}{8}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).

  The midpoint is $c = \frac{a+b}{2} = \frac{21}{16}$. We compute $f(c) = \frac{1069}{4096} > 0$.

  Hence, $[a,c] = \left[\frac{5}{4}, \frac{21}{16}\right]$ must contain a root of $f(x)$.

After 4 steps of the bisection method, we know that $\sqrt[3]{2}$ must lie in the interval $\left[\frac{5}{4}, \frac{21}{16}\right] = [1.25, 1.3125]$.

Comment. For comparison, $\sqrt[3]{2} \approx 1.2599$. Note that our approximations are a bit more impressive if we think in terms of binary digits. Then each step provides one additional digit of accuracy (because the length of the interval of uncertainty is cut by half).

Comment. The above steps are on purpose written in a repetitive manner (reusing the same variable names a, b, c with new values) to make it easier to translate the process into Python code.

Example 34. (continued) We wish to determine an approximation for $\sqrt[3]{2}$ by applying the bisection method to the function $f(x) = x^3 - 2$ on the interval $[1,2]$.

(a) After how many iterations of bisection will the final interval have size less than 10^{-6}?

(b) If we approximate $\sqrt[3]{2}$ using the midpoint of the final interval, how many iterations of bisection do we need to guarantee that the (absolute) error is less than 10^{-6}?

Solution.

(a) At each iteration, the width of the interval is divided by 2. Hence, the width of the interval after n steps will be exactly $\frac{2-1}{2^n} = \frac{1}{2^n}$. Solving $\frac{1}{2^n} < 10^{-6}$ for n, we find $n > -\log_2(10^{-6}) = 6\log_2(10) \approx 19.93$. Hence, we need 20 iterations.

(b) Again, the width of the interval after n steps will be exactly $\ell = \frac{2-1}{2^n} = \frac{1}{2^n}$. Since $\sqrt[3]{2}$ is contained in this interval, the absolute error of approximating it with the midpoint is at most $\ell/2 = \frac{1}{2^{n+1}}$. Solving $\frac{1}{2^{n+1}} < 10^{-6}$ for n, we find $n > -\log_2(10^{-6}) - 1 = 6\log_2(10) - 1 \approx 18.93$. Hence, we need 19 iterations.

Comment. We didn't refer to the first part on purpose. Given the answer 20 from the first part, can you see that the answer must be $20 - 1 = 19$?

If we use bisection to compute a root of a (continuous) function $f(x)$ on $[a,b]$, then:

- After n iterations, the (absolute) error is less than $\frac{b-a}{2^{n+1}}$.

This assumes that we approximate the root using the midpoint of the final interval.

Important comment. This error bound is independent of $f(x)$.

- Each additional iteration requires 1 function evaluation.

Python: Implementing the bisection method

Example 35. Recall that the bisection method cuts an interval $[a,b]$ into two halves by computing the midpoint $c=\frac{a+b}{2}$. It then chooses either $[a,c]$ or $[c,a]$ as the improved new interval.

Give a simple condition for when the interval $[a,c]$ is chosen.

Solution. We choose $[a,c]$ if $f(a)$ and $f(c)$ have opposite signs.

This is equivalent to $f(a)f(c)<0$.

Comment. Here, and in the sequel, we will be very nonchalant about the possibility that $f(c)=0$. This would mean that we have accidentally found the actual root. This is not something that we expect to happen in most applications (though serious code would have to consider the case where $f(c)$ is 0 to within the precision we are working with). Note that if $f(c)=0$ then our above rule would always choose the right half of the interval (and that would be fine).

Comment. In Python, we can therefore implement an iteration of bisection as follows:

```
>>> # we start with the interval [a, b]
    c = (a + b) / 2
    if f(a)*f(c) < 0:
        b = c  # pick [a, c] as the next interval
    else:
        a = c  # pick [c, b] as the next interval
```

Even if you have never used/seen such an if-then-else statement before, the above code can be read almost as an English sentence. Note how we indent things after `if` and after `else` (the `else` part can be omitted if we only want to do something if a condition is true) to group the code that we want to be executed in each case.

Advanced comment. One potential issue with using the product $f(a)f(c)$ rather than directly comparing the signs is that, depending on our available precision, $f(a)f(c)$ might run into an underflow (note that $f(a)$ and $f(c)$ are each getting smaller by construction) and be treated as zero. Here is a simple artificial example illustrating the issue:

```
>>> def f(x):
        return (x-1)**99

>>> f(0.99) < 0

  True

>>> f(1.01) > 0

  True

>>> f(0.99) * f(1.01) < 0

  False
```

With the representation of floats and their precision in mind, explain the above output!

When writing serious code, we therefore have to account for this and should instead compare the sign of $f(a)$ to the sign of $f(b)$ (and not compute the product). We will ignore this issue in the sequel.