Notes for Lecture 7

Mon, 9/18/2023

Example 40. In Example 38, which we continued in the previous example, only the left point ever got updated. Will that always be the case? Explain!

Solution. For $f(x) = x^3 - 2$, we have $f'(x) = 3x^2$ and $f''(x) = 6x$.

Therefore we have $f'(x) > 0$ as well as $f''(x) > 0$ for all $x \in [1, 2]$. This means that our function is increasing as well as concave up (on the interval $[1, 2]$).

Because it is concave up, its graph will always lie below the secant lines we construct (see below).

Combined with the function being increasing, the regula falsi points (roots of the secant lines) will always be to the left of the true root (here $\sqrt[3]{2}$).

Accordingly, the right endpoint will never get updated.

Review of concavity. Recall that a function $f(x)$ is **concave up** (like any part of a parabola opening upward) at $x = c$ if $f''(c) > 0$. At such a point, the graph of the function lies above the tangent line (at least locally). Make a sketch! On the other hand, this means that for sufficiently small intervals $[a, b]$ around c, the graph will lie below the secant line through $(a, f(a))$ and $(b, f(b))$.

Let us note the following differences between bisection and regula falsi:

- The intervals produced by bisection shrink by a factor of $1/2$ per iteration. On the other hand, the intervals produced by regula falsi usually don't drop below a certain length.

  For instance. This is illustrated by Example 38. In that case, the generated intervals are all of the form $[a, 2]$ where the left side a approaches $\sqrt[3]{2}$ from below. In particular, these intervals will always have length larger than $2 - \sqrt[3]{2} \approx 0.74$.

- Despite this, the sequence c_n of "new" interval endpoints produced by regula falsi can be shown to always converge to a root. Often the approximations c_n converge faster than the approximations obtained through bisection, but it can also be the other way around.

  Comment. There are, however, variations of regula falsi that are more reliably faster than bisection.

- Bisection is guaranteed to converge to a root at a certain rate (namely, one bit per iteration). Regula falsi frequently but not always converges faster, but we cannot guarantee a certain rate (this depends on the involved function $f(x)$).

Example 41. Suppose we use bisection or regula falsi to compute a root of some function. Several iterations result in the intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$. Based on these intervals, what is our approximation of the root?

(a) In the case of bisection.

(b) In the case of regula falsi.

Solution.

(a) In the case of bisection, the generically best choice for our approximation is the midpoint of the final interval $(a_n + b_n)/2$.

(b) In the case of regula falsi, our approximation is the "new" endpoint of the final interval. More precisely, the approximation is a_n if $b_n = b_{n-1}$ and it is b_n if $a_n = a_{n-1}$.

Secant method

The **secant method** for computing a root of a function $f(x)$ is a modification of regula falsi where we do not try to bracket the root (in other words, we do not produce intervals that are guaranteed to contain the root).

Instead, starting with two initial approximations x_0 and x_1, we construct $x_2, x_3, \ldots$ by the rule

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

Comment. In other words, if $a = x_{n-1}$ and $b = x_n$ are the two most recent approximations, then the next approximation is

$$x_{n+1} = c = \frac{a f(b) - b f(a)}{f(b) - f(a)},$$

and, as in regula falsi, this is the root of the secant line through $(a, f(a))$ and $(b, f(b))$. While regula falsi next determines whether to continue with the interval $[a, c]$ or with $[c, b]$, the secant method always continues with b and c as the next pair of approximations (in particular, the root does not need to lie between b and c).

Advanced comment. The formula for x_{n+1} is somewhat problematic because it is prone to round-off errors. Namely, if x_n converges to a root of $f(x)$, then in both the numerator and denominator of that formula we are subtracting numbers of almost equal value. This can result in damaging loss of precision.

Why is this not an issue for regula falsi? (Hint: What do we know about the signs of $f(a)$ and $f(b)$?)

Example 42. Determine an approximation for $\sqrt[3]{2}$ by applying the secant method to the function $f(x) = x^3 - 2$ with initial approximations $x_0 = 1$ and $x_1 = 2$. Perform 3 steps.

Solution.

- $x_2 = \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)} = \frac{8}{7} \approx 1.1429$
- $x_3 = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)} = \frac{75}{62} \approx 1.2097$
- $x_4 = \frac{x_2 f(x_3) - x_3 f(x_2)}{f(x_3) - f(x_2)} = \frac{989{,}312}{782{,}041} \approx 1.2650$

After 3 steps of the secant method, our approximation for $\sqrt[3]{2}$ is $\frac{989{,}312}{782{,}041} \approx 1.2650$.

Comment. For comparison, $\sqrt[3]{2} \approx 1.2599$.

Comment. Note that the interval $[x_2, x_3]$ does not contain the root $\sqrt[3]{2}$.

Compare Example 42 to Example 38 where we used regula falsi instead (note how the first two iterations resulted in the same approximations). In operational terms, the secant method is a simpler version of regula falsi since we are not trying to determine an interval that is guaranteed to contain a root.

It may therefore come as a surprise that the secant method typically converges considerably faster than regula falsi. However, we no longer have a guarantee of convergence (and the situation in general depends on the initial approximations as well as the function $f(x)$).

Example 43. Python The following is code for performing a fixed number of iterations of the secant method. Note that the code is a simplified version of our code in Example 39 for the regula falsi method.

```
>>> def secant_method(f, a, b, nr_steps):
        for i in range(nr_steps):
            c = (a*f(b) - b*f(a)) / (f(b) - f(a))
            a = b
            b = c
        return b
```

Comment. This time, we return only a single value which is an approximation to the desired root. (Recall that the secant method does not provide intervals containing the true root.)

As before, let us use this code to automatically perform the computations from Example 42.

```
>>> def my_f(x):
        return x**3 - 2

>>> from fractions import Fraction

>>> [secant_method(my_f, Fraction(1), Fraction(2), n) for n in range(1,4)]

   [Fraction(8, 7), Fraction(75, 62), Fraction(989312, 782041)]
```

Newton's method

The **Newton method** proceeds as the secant method, except that it uses tangents instead of secants. In particular, instead of two previous points x_{n-1}, x_n (so that we construct a secant line) we only require a single point x_n to compute the next point.

Example 44. Derive a formula for the root of the tangent line through $(a, f(a))$.

Solution. The line has slope $m = f'(a)$. (We could also pick the other point.)
Since it passes through $(a, f(a))$, the line has the equation $y - f(a) = m(x - a)$.
To find its root (or x-intercept), we set $y = 0$ and solve for x.
We find that the root is $x = a - \frac{f(a)}{m} = a - \frac{f(a)}{f'(a)}$.
Comment. Compare the above derivation with what we did for the regula falsi method.

Thus, given an initial approximation x_0, the Newton method constructs $x_1, x_2, \ldots$ by the rule

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

Comment. In contrast to the secant method, the Newton method requires us to be able to compute $f'(x)$. Also, per iteration we need two function evaluations (one for f and one for f') whereas the secant method only requires a single function evaluation.
Comment. If we use the approximation $f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$ (which is a good approximation if x_{n-1} and x_n are sufficiently close) in the Newton method, then we actually obtain the secant method.

Example 45. Determine an approximation for $\sqrt[3]{2}$ by applying Newton's method to the function $f(x)=x^3-2$ with initial approximation $x_0=1$. Perform 3 steps.

Solution. We compute that $f'(x)=3x^2$.

- $x_1=x_0-\frac{f(x_0)}{f'(x_0)}=\frac{4}{3}\approx 1.3333$
- $x_2=x_1-\frac{f(x_1)}{f'(x_1)}=\frac{91}{72}\approx 1.2639$
- $x_3=x_2-\frac{f(x_2)}{f'(x_2)}=\frac{1{,}126{,}819}{894{,}348}\approx 1.2599$

After 3 steps of Newton's method, our approximation for $\sqrt[3]{2}$ is $\frac{1{,}126{,}819}{894{,}348}\approx 1.259933$.

Comment. For comparison, $\sqrt[3]{2}\approx 1.259921$. The error is only 0.000012!

After one more iteration, the error drops to an astonishing $1.2\cdot 10^{-10}$.

After one more iteration, the error drops to an astonishing $1.2\cdot 10^{-20}$.

It looks like the number of correct digits is doubling at each step!!

We will soon prove that this is indeed the case.

Example 46. Python The following code implements the Newton method. Note that we need as input both a function `f` and its derivative `fd`:

```
>>> def newton_method(f, fd, x0, nr_steps):
        for i in range(nr_steps):
            x0 = x0 - f(x0)/fd(x0)
        return x0
```

Let us use this code to automatically perform the computations from Example 45.

```
>>> def f(x):
        return x**3 - 2

>>> def fd(x):
        return 3*x**2

>>> [newton_method(f, fd, 1, n) for n in range(1,5)]

    [1.3333333333333333, 1.2638888888888888, 1.259933493449977, 1.2599210500177698]
```

Next, let us compare these values to $\sqrt[3]{2}$, the actual root of $f(x)$.

```
>>> [newton_method(f, fd, 1, n) - 2**(1/3) for n in range(1,6)]

    [0.07341228343846007, 0.003967838994015649, 1.24435551038804e-05,
    1.2289658180009155e-10, 0.0]
```

It really looks like the number of correct digits is doubling at each step! Can you explain what happened for the last output for which we got `0.0`?

Well, we expect about 20 correct decimal digits (translating to about $20\cdot\log_2 10\approx 66.4$ binary digits). That is more than the number of significant digits that can be stored in a double-precision float. Accordingly, the error is going to be rounded down to 0.

Finally, for comparison with Example 45, here are the first few exact values:

```
>>> from fractions import Fraction

>>> [newton_method(f, fd, Fraction(1), n) for n in range(1,4)]

    [Fraction(4, 3), Fraction(91, 72), Fraction(1126819, 894348)]
```