Example 76. Python The following code implements the Newton method specifically for computing a root of $f(x) = (x-r)(x-1)(x+2)$ as in the previous example.

```
>>> def newton_f(r, x, nr_steps):
        for i in range(nr_steps):
            x = x - ((x-r)*(x-1)*(x+2))/(3*x**2-2*(r-1)*x-r-2)
        return x
```

We then write a function to tell us the how close the result of Newton's method is to $x^* = 1$ (the root that we are trying to compute). Namely, `newton_f_cb_1` will return the number of correct digits in base 2.

```
>>> from math import log2

>>> def newton_f_cb_1(r, x, nr_steps):
        return -log2(abs(1 - newton_f(r, x, nr_steps)))
```

Here is the typical behaviour which we get if $r \neq 1$ and $r \neq 4$. We chose $r = 2$ and for the initial approximation we chose $x_0 = 0.4$. First, we list the result of Newton's method and observe that the approximations are indeed approaching 1 (recall that we are only guaranteed convergence if x_0 is close enough to 1). We then list the number of correct bits for those approximations:

```
>>> [newton_f(2, 0.4, n) for n in range(1,5)]

   [0.9333333333333332, 0.9974499089253187, 0.9999956903710115, 0.9999999999876182]

>>> [newton_f_cb_1(2, 0.4, n) for n in range(1,5)]

   [3.9068905956085165, 8.615235511834927, 17.824004894803025, 36.2329923774517]
```

Observe how the number of correct digits indeed roughly doubles.

Next, we likewise consider the problematic case $r = 1$:

```
>>> [newton_f(1, 0.4, n) for n in range(1,5)]

   [0.7428571428571429, 0.877751756440281, 0.9402023433223725, 0.9704083354780979]

>>> [newton_f_cb_1(1, 0.4, n) for n in range(1,5)]

   [1.9593580155026542, 3.032114357937968, 4.063767239896592, 5.078665339814252]
```

Observe how the number of correct digits no longer doubles. Instead it roughly increases by 1 per iteration, exactly as we had predicted.

Finally, we consider the exceptionally good case $r = 4$:

```
>>> [newton_f(4, 0.4, n) for n in range(1,5)]

   [1.0545454545454547, 0.9999639010889838, 1.0000000000000104, 1.0]

>>> [newton_f_cb_1(4, 0.4, n) for n in range(1,4)]

   [4.1963972128035, 14.757685157968053, 46.445411148322364]
```

Observe how the number of correct digits now roughly triples, in accordance with our prediction.

Comparison of root finding algorithms

Now that we have seen several root finding algorithms, which one is the best?

Well, it really depends on the situation. Below are some of the differences between the methods.

In practice, one often uses hybrid algorithms that combine several methods.

All methods require a continuous function.

- Bisection
  - each iteration is guaranteed to provide a correct binary digit; no other method can guarantee this for all functions
  - requires an initial interval containing a root such that the function values at the endpoints have opposite signs (in particular, does not work for double roots (or any even order roots)); on the other hand, it provides a guaranteed interval containing the root
  - no requirement on $f(x)$ besides continuity; for the other methods, the performance depends on $f(x)$
  - essentially linear convergence with rate $\frac{1}{2}$
- Regula falsi
  - also requires an initial interval containing a root like bisection
  - one endpoint of the interval typically gets stuck
  - rarely used directly, but rather in its improved forms, such as the Illinois method
  - always converges, typically linearly with variable rate
- Illinois method (see next pages for bonus material!)
  - improved version of regula falsi
  - the interval now shrinks to root
  - always converges, typically with order $\sqrt[3]{3} \approx 1.442$
- Secant method
  - only requires an initial approximation
  - only converges if initial approximation is good enough
  - potential numerical issues due to loss of precision in near zero denominator
  - typical order of convergence $\phi = (1+\sqrt{5})/2 \approx 1.618$
- Newton's method
  - similar to secant method
  - requires derivative
  - extends well to other contexts such as approximating functions or power series rather than numbers
  - typical order of convergence 2
  - however, adjusted for two function evaluations ($f(x)$ and $f'(x)$), order of convergence $\sqrt{2} \approx 1.414$

Bonus material: The Illinois method

Example 77. Python In Example 41 we implemented the regula falsi method. As we have observed, a weakness of this method is that we typically end up only updating one endpoint of the interval. The **Illinois algorithm** is an extension of the regula falsi method that works to remedy this issue.

Recall that the regula falsi method uses $c = \frac{a f_b - b f_a}{f_b - f_a}$ with $f_a = f(a)$ and $f_b = f(b)$ to cut each interval $[a,b]$ into the two parts $[a,c]$ and $[c,b]$.

The Illinois algorithm proceeds likewise but, after an endpoint has been retained for a second time, the corresponding value f_a or f_b is replaced with half its value. In other words, if a was not updated in this or the previous step, then f_a (to be used in the next iteration) is replaced with $f_a/2$; likewise, if b was not updated in this or the previous step, then f_b is replaced with $f_b/2$.

```
# start with an interval [a,b]
fa = f(a)
fb = f(b)
repeat
  # compute the regula falsi point
  c = (a*fb - b*fa) / (fb - fa)
  fc = f(c)
  # set new interval [a,b] according to signs of f
  ...
  if left endpoint was also updated the previous time
    fb = fb/2
  if right endpoint was also updated the previous time
    fa = fa/2
```

Can you complete this pseudo-implementation? Here is one approach that we can take:

- Start with the code that we wrote in class for the regula falsi method.
- Adjust that code (like we did for the bisection method) to only use one function evaluation per iteration. Do that by introducing variables `fa`, `fb`, `fc` for the values of $f(x)$ at $x = a$, b, c.
- Add a new variable to your code that keeps track of whether we most recently changed the left or the right endpoint of the interval. You can, for instance, define a variable `updated_endpoint` that is initially set to 0, and which is set to 1 after the right endpoint is updated and to -1 after the left endpoint is updated.

  That way, if we are about to update, say, the left endpoint, then we can test whether `updated_endpoint` is -1 as that would tell us that we are now updating the left endpoint for a second time in a row. In that case, we set `fb = fb/2`.

Advanced comment. There are other, more clever, approaches to implementing the Illinois method. For instance, one could stop making a and b the left and right endpoints of the interval and, instead, always make b the newly added endpoint; then one can test whether we repeatedly change the same endpoint by looking at the signs of the corresponding values of f. This is done by M. Dowell and P. Jarratt in [*A Modified Regula Falsi Method for Computing the Root of an Equation*, 1971], where they describe and analyze the Illinois method. As a very minor point, their implementation might proceed slightly different from ours because we start with an interval $[a,b]$ whereas their implementation thinks of a and b as two approximations, with b being the more "recent" one (accordingly, their implementation might divide f_a by 2 already at the end of the first iteration).

Let us revisit the computations we did in Example 40 but with the regula falsi method updated to the Illinois algorithm. The first two iterations should result in the same intervals:

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

>>> from fractions import Fraction

>>> illinois(my_f, Fraction(1), Fraction(2), 1)

   [Fraction(8, 7), Fraction(2, 1)]

>>> illinois(my_f, Fraction(1), Fraction(2), 2)

   [Fraction(75, 62), Fraction(2, 1)]
```

However, at the end of the second iteration (since the right endpoint has not changed in this or the previous iteration), $f_b = 6$ (since $f(2) = 6$) is replaced with $f_b = 3$. As a result, in the third iteration, we end up replacing the right endpoint:

```
>>> illinois(my_f, Fraction(1), Fraction(2), 3)

   [Fraction(75, 62), Fraction(974462, 769765)]
```

For further testing, in the next two iterations we replace the left endpoints (since the fractions are becoming large, we are using floats below; note that the first command just repeats the above computation with floats):

```
>>> illinois(my_f, 1, 2, 3)

   [1.2096774193548387, 1.2659214175754938]

>>> illinois(my_f, 1, 2, 4)

   [1.2596760796087871, 1.2659214175754938]

>>> illinois(my_f, 1, 2, 5)

   [1.2599198867703156, 1.2659214175754938]
```

Consequently, at the end of the fifth iteration, the value of f_b (which is $f(1.2659\ldots)$) is again replaced by half its value. Once more, this results in b being updated in the next iteration:

```
>>> illinois(my_f, 1, 2, 6)

   [1.2599198867703156, 1.2599222015292841]
```