Applying fixed-point iteration directly

Note that any equation $f(x)=0$ can be rewritten in many ways as a fixed-point equation $g(x)=x$.

For instance. We can always rewrite $f(x)=0$ as $f(x)+x=x$ (i.e. choose $g(x)=f(x)+x$).

We can then attempt to find a root x^* of $f(x)$ by fixed-point iteration on $g(x)$.

In other words, we start with a value x_0 (an initial approximation) and then compute $x_1, x_2, \ldots$ via $x_{n+1}=g(x_n)$.

Theorem 65 tells us whether that such a fixed-point iteration on $g(x)$ will locally converge to x^*. Moreover, it tells us the order of convergence.

Example 68. Suppose we are interested in computing the roots of $x^2-x-1=0$.

The roots are the golden ratio $\phi=\frac{1}{2}(1+\sqrt{5})\approx 1.618$ and $\psi=\frac{1}{2}(1-\sqrt{5})\approx -0.618$.

There are many ways to rewrite this equation as a fixed-point equation $g(x)=x$. The following are three possibilities:

(a) Rewrite as $x=x^2-1$, so that $g(x)=x^2-1$.

(b) Rewrite first as $x^2=x+1$ and then as $x=1+\frac{1}{x}$, so that $g(x)=1+\frac{1}{x}$.

(c) Rewrite first as $\underbrace{x^2-x}_{=x(x-1)}=1$ and then as $x=\frac{1}{x-1}$, so that $g(x)=\frac{1}{x-1}$.

In each of these three cases and for each root, decide whether fixed-point iteration converges. If it does, determine the order and rate of convergence.

Solution.

(a) In this case, we have $g(x)=x^2-1$ and $g'(x)=2x$.
Since $|g'(\phi)|\approx 3.236>1$ as well as $|g'(\psi)|\approx 1.236>1$, fixed-point iteration does not converge locally to either root.

(b) In this case, we have $g(x)=1+\frac{1}{x}$ and $g'(x)=-\frac{1}{x^2}$.
Since $|g'(\phi)|=\frac{1}{\phi+1}\approx 0.382<1$ and $|g'(\psi)|=\phi+1\approx 2.618>1$, fixed-point iteration converges locally to ϕ but does not converge locally to ψ. Moreover, the convergence to ϕ is linear with rate 0.382.

(c) In this case, we have $g(x)=\frac{1}{x-1}$ and $g'(x)=-\frac{1}{(x-1)^2}$.
Since $|g'(\phi)|=\phi+1\approx 2.618>1$ and $|g'(\psi)|=\frac{1}{\phi+1}\approx 0.382<1$, fixed-point iteration converges locally to ψ but does not converge locally to ϕ. Moreover, the convergence to ψ is linear with rate 0.382.

Armin Straub
straub@southalabama.edu

Order of convergence of Newton's method

Recall that computing a root x^* of $f(x)$ using Newton's method is equivalent to fixed-point iteration of $g(x) = x - \frac{f(x)}{f'(x)}$.

Comment. In each case, we start with x_0 and iteratively compute $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Theorem 69. Suppose that f is twice continuously differentiable and $f(x^*) = 0$.

- **(typical case)** Newton's method (locally) converges to x^* quadratically with rate $\frac{1}{2}|f''(x^*)/f'(x^*)|$ provided that $f'(x^*) \neq 0$.
- **(troubled case)** If $f'(x^*) = 0$, then Newton's method either does not converge at all or it converges linearly.

Note that, if $f(x^*) = 0$ and $f'(x^*) = 0$, then x^* is a repeated root of $f(x)$. We thus conclude that Newton's method is troubled if we are trying to compute a repeated root.

- **(exceptionally good case)** If $f'(x^*) \neq 0$ and $f''(x^*) = 0$, then Newton's method even converges with order at least 3.

Important comment. In short, Newton's method typically converges quadratically (though in very special cases it can converge even faster) except in the case of repeated roots.

Proof. We apply Theorem 65 to analyze the fixed-point iteration of $g(x) = x - \dfrac{f(x)}{f'(x)}$.

Using the quotient rule we compute that

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}.$$

If $f(x^*) = 0$ and $f'(x^*) \neq 0$, then we have $g'(x^*) = 0$. By Theorem 65 this implies that fixed-point iteration converges at least quadratically.

To determine the rate of convergence, we further compute (again using the quotient and product rule) that

$$g''(x) = \frac{(f'(x)f''(x) + f(x)f'''(x))f'(x)^2 - 2f(x)f''(x)f'(x)f''(x)}{f'(x)^4}.$$

From this (unsimplified) expression and $f(x^*) = 0$ we conclude that $g''(x^*) = \dfrac{f''(x^*)}{f'(x^*)}$.

By Theorem 65 this implies that the convergence is quadratic with rate $\dfrac{1}{2}\left|\dfrac{f''(x^*)}{f'(x^*)}\right|$.

Moreover, if $f''(x^*) = 0$ then $g''(x^*) = 0$ so that the convergence is cubic (or higher). □

Example 70. (cont'd) Does Newton's method applied to finding a root of $f(x) = x^3 - 2$ converge locally to $\sqrt[3]{2}$? If so, determine the order and the rate.

This is a continuation of Examples 60 and 67.

Solution. Recall that Newton's method typically converges to x^* with order 2 and rate $\frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right|$.

With $x^* = \sqrt[3]{2}$, we compute $f'(x^*) = 3(x^*)^2 = 3 \cdot 2^{2/3}$. Since $f'(x^*) \neq 0$, we already know that Newton's method converges at least with order 2.

We further compute $f''(x^*) = 6x^* = 6 \cdot 2^{1/3} = 3 \cdot 2^{4/3}$. Since $f''(x^*) \neq 0$, we know that Newton's method does not converge with order larger than 2.

Therefore, Newton's method converges to $x^* = \sqrt[3]{2}$ with order 2 and rate $\frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| = \frac{1}{2}\frac{3 \cdot 2^{4/3}}{3 \cdot 2^{2/3}} = 2^{-1/3} \approx 0.7937$.

Of course, this matches what we computed in Example 67.

Armin Straub
straub@southalabama.edu

Example 71. $f(x) = e^{-x} - x$ has the unique root $x^* \approx 0.567$. Determine whether Newton's method converges locally to x^*. If it does, what is the order and rate of convergence?

Solution. We compute that $f'(x) = -e^{-x} - 1$ and $f''(x) = e^{-x}$.

Since $x^* = e^{-x^*}$, we have $f'(x^*) = -x^* - 1 \neq 0$.

Hence, by Theorem 69, Newton's method converges to x^* quadratically.

Moreover, the rate is $\frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| = \frac{1}{2}\left|\frac{e^{-x^*}}{-e^{-x^*} - 1}\right| = \frac{1}{2}\left|\frac{x^*}{-x^* - 1}\right| \approx 0.181$.

Review. If $f(x^*) = 0$ and $f'(x^*) \neq 0$, then Newton's method (locally) converges to x^* quadratically with rate $\frac{1}{2}|f''(x^*) / f'(x^*)|$.

Note that we can see from here that $f'(x^*) = 0$ is problematic; indeed, in that case, we don't get quadratic convergence (but rather divergence or linear convergence).

We can also see that, if $f''(x^*) = 0$, then we should get even better convergence; indeed, in that case, we get cubic convergence or better.

Example 72. Consider $f(x) = (x - r)(x - 1)(x + 2)$ where r is some constant. Suppose we want to use Newton's method to calculate the root $x^* = 1$.

(a) For which values of r is Newton's method guaranteed to converge (at least) quadratically to $x^* = 1$?

(b) Analyze the cases in which Newton's method does not converge quadratically to $x^* = 1$. Does it still converge? If so, what can we say about the order and rate of convergence?

(c) For which values of r does Newton's method converge to $x^* = 1$ faster than quadratically?

Solution.

(a) We have $f(x) = x^3 - (r - 1)x^2 - (r + 2)x + 2r$ and, hence, $f'(x) = 3x^2 - 2(r - 1)x - (r + 2)$.

Note that $f'(1) = 3 - 3r = 0$ if and only if $r = 1$.

Theorem 69 implies that Newton's method converges (at least) quadratically to $x^* = 1$ if $r \neq 1$.

Comment. Note that $r = 1$ is precisely the case where 1 becomes a double root of $f(x)$.

(b) We need to analyze the case $r = 1$.

In that case $f(x) = (x - 1)^2(x + 2)$ and $f'(x) = 3x^2 - 3 = 3(x - 1)(x + 1)$.

Newton's method applied to $f(x)$ is equivalent to fixed-point iteration of

$$g(x) = x - \frac{f(x)}{f'(x)} = x - \frac{(x - 1)^2(x + 2)}{3(x - 1)(x + 1)} = x - \frac{x^2 + x - 2}{3(x + 1)} = \frac{2}{3}x + \frac{2}{3}\frac{1}{x + 1}.$$

We compute that $g'(x) = \frac{2}{3} - \frac{2}{3}\frac{1}{(x + 1)^2}$ so that, in particular, $g'(1) = \frac{2}{3} - \frac{2}{3}\frac{1}{4} = \frac{1}{2}$.

Since $0 \neq |g'(1)| < 1$ we conclude, by Theorem 65, that Newton's method (locally) converges to $x^* = 1$. Moreover, the convergence is linear with rate $\frac{1}{2}$.

Comment. Since $\frac{1}{2} = 2^{-1}$, this means that we gain roughly one correct binary digit per iteration.

(c) We continue the calculation from the first part. According to Theorem 69, Newton's method converges to 1 faster than quadratic if $f'(1) \neq 0$ and $f''(1) = 0$.

We calculate $f''(x) = 6x - 2(r - 1)$. Thus $f''(1) = 8 - 2r = 0$ if and only if $r = 4$.

Hence, Newton's method converges to 1 faster than quadratic if $r = 4$.

Important comment. Note that what we are observing is exactly as what we should expect: Newton's method typically converges quadratically (though in very special cases it can converge even faster; here, $r = 4$) except in the case of repeated roots (here, $r = 1$).

Example 73. `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.

Armin Straub
straub@southalabama.edu

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 74. Python In Example 39 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 38 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 $f(1.2659...)$) 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]
```