**Example 39.** 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 35 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 38.

```
>>> 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 40.** 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.

Armin Straub
straub@southalabama.edu

**Example 41.** 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 $f'(x)=3x^2$.

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

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 42.** Python The following code implements the Newton method specifically for computing a root of $f(x)=x^3-2$ as in Example 41 (cr2 is meant to be short for cube root of 2).

```
>>> def newton_cr2(x0, nr_steps):
        for i in range(nr_steps):
            x0 = x0 - (x0**3-2)/(3*x0**2)
        return x0
```

Let us use this code to automatically perform the computations from Example 41.

```
>>> from fractions import Fraction
>>> [newton_cr2(Fraction(1), n) for n in range(1,4)]
   [Fraction(4, 3), Fraction(91, 72), Fraction(1126819, 894348)]
```

Next, let us compare these values to $\sqrt[3]{2}$, the actual root of $f(x)$. Note that, if $x$ is a (close) approximation of $\sqrt[3]{2}$, then the number of correct binary digits of $x$ is $\lfloor -\log_2|x-\sqrt[3]{2}|\rfloor$. The following function uses this to compute the correct number of bits after a certain number of steps of the Newton method:

```
>>> from math import log2
>>> def newton_cr2_correctbits(x0, nr_steps):
        return -log2(abs(2**(1/3) - newton_cr2(x0, nr_steps)))
>>> [newton_cr2_correctbits(Fraction(1), n) for n in range(1,5)]
   [3.7678347129038254, 7.977430799070329, 16.294241754402005, 32.92183615918938]
```

**Comment.** What about the number of correct bits after $5$ steps (one more step)? If you run our code above, you will receive an error (`ValueError: math domain error`). Can you explain why?

Well, we expect about $64$ correct 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 $0$. We then run into trouble because we ask for the logarithm of $0$ (which is a singularity).

**Example 43.** Apply Newton's method to $g(x)=x^3-2x+2$ and initial value $x_0=0$.

**Solution.** Using $g'(x)=3x^2-2$, we compute that $x_1=x_0-\frac{g(x_0)}{g'(x_0)}=1$, $x_2=x_1-\frac{g(x_1)}{g'(x_1)}=1-\frac{1}{1}=0$.

Since $x_2=x_0$, the Newton method will now repeat and we are stuck in a $2$-cycle.

In particular, the Newton method does not converge in this case.

**Comment.** It is possible to run into $n$-cycles for larger $n$ as well when doing Newton iterations (for instance, try $f(x)=x^5-x-1$ and initial value $x_0=0$). When computing numerically, it is not particularly likely that we will run into a perfect cycle. However, such cycles can be **attractive**. Meaning that we get closer and closer to the cycle if we start with a nearby point. This is illustrated by the Python code experiment below.

**Example 44.** Python The following code implements the Newton method specifically for computing a root of $g(x)=x^3-2x+2$ as in the previous example.

```
>>> def newton_g(x0, nr_steps):
        for i in range(nr_steps):
            x0 = x0 - (x0**3-2*x0+2)/(3*x0**2-2)
        return x0
```

The following confirms that we have a $2$-cycle starting with $0$:

```
>>> [newton_g(0, n) for n in range(8)]

   [0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
```

On the other hand, this is what happens if we start with a point close to $0$:

```
>>> [newton_g(0.1, n) for n in range(8)]

   [0.1, 1.0142131979695432, 0.07965576631987636, 1.0090987403727651, 0.05222652653371296,
   1.0039651847274838, 0.02332943565497303, 1.0008043531824031]
```

Notice how we are being attracted by the $2$-cycle.

## Review: Taylor series

Recall from Calculus that, if $f(x)$ is **analytic** around $x=c$, then

$$f(x) \quad = \quad \sum_{n=0}^{\infty} \frac{f^{(n)}(c)}{n!}(x-c)^n = f(c)+f'(c)(x-c)+\frac{1}{2}f''(c)(x-c)^2+\ldots$$

The series on the right-hand side is called the **Taylor series** of $f(x)$ around $x=c$.

**Advanced comment.** We only get the equality between $f(x)$ and its Taylor series for functions that are **analytic** at $x=c$ (by definition, these are functions than can be expanded as a power series; the above makes it explicit what the coefficients of that power series have to be). Fortunately, all elementary functions (the ones we can express as algebraic expressions with exponentials, logarithms and trig functions) are analytic at almost all points.

On the other hand, for instance, the Taylor series for the function $f(x)=e^{-1/x^2}$ at $x=0$ is zero (because all derivatives of $f(x)$ are zero for $x=0$) while $f(x)$ is not zero (however, note that $x=0$ is clearly a problematic point of the formula for $f(x)$; that function is analytic at all other points). Since $f(x)$ is infinitely differentiable, this illustrates that being analytic is a stronger property than being infinitely differentiable. For other functions, it is possible that the Taylor series might not converge at all.

**Comment.** The Taylor series of $f(x)$ around $x=0$ takes the form $f(0)+f'(0)x+\frac{1}{2}f''(0)x^2+\ldots$. In practice, we can often shift things so that the Taylor series of interest are around $x=0$.

**Example 45.** Determine the Taylor series of $e^x$ around $0$.

**Solution.** If $f(x)=e^x$, then $f^{(n)}(x)=e^x$. In particular, we have $f^{(n)}(0)=1$ for all $n$.

Consequently, we have $e^x=\sum_{n=0}^{\infty}\frac{x^n}{n!}=1+x+\frac{1}{2}x^2+\frac{1}{6}x^3+\ldots$

**Comment.** $e^x$ is analytic everywhere and so it equals its Taylor series.

## Python assignment #2

**Example 46.** `Python` In Example 35 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
```

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 34 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]
```