

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 46. 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 $x_1, x_2, \dots$ 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 47. 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 48. 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 47.

```
>>> 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 47, 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)]
```

Example 49. 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. $g(x)$ has one real root at $x \approx -1.7693$ (as well as two complex roots). Make a plot of $g(x)$!

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 50. Python We apply the Newton method from our previous example to computing a root of $g(x) = x^3 - 2x + 2$. For that, we define g and its derivative as functions in Python:

```
>>> def g(x):  
    return x**3 - 2*x + 2  
  
>>> def gd(x):  
    return 3*x**2 - 2
```

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

```
>>> [newton_method(g, gd, 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_method(g, gd, 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.