

Python: Implementing the bisection method
---

**Example 33.** Recall that the bisection method cuts an interval  $[a, b]$  into two halves by computing the midpoint  $c = \frac{a+b}{2}$ . It then chooses either  $[a, c]$  or  $[c, a]$  as the improved new interval.

Give a simple condition for when the interval  $[a, c]$  is chosen.

**Solution.** We choose  $[a, c]$  if  $f(a)$  and  $f(c)$  have opposite signs.

This is equivalent to  $f(a)f(c) < 0$ .

**Comment.** Here, and in the sequel, we will be very nonchalant about the possibility that  $f(c) = 0$ . This would mean that we have accidentally found the actual root. This is not something that we expect to happen in most applications (though serious code would have to consider the case where  $f(c)$  is 0 to within the precision we are working with). Note that if  $f(c) = 0$  then our above rule would always choose the right half of the interval (and that would be fine).

**Comment.** In Python, we can therefore implement an iteration of bisection as follows:

```
>>> # we start with the interval [a, b]
    c = (a + b) / 2
    if f(a)*f(c) < 0:
        b = c # pick [a, c] as the next interval
    else:
        a = c # pick [c, b] as the next interval
```

Even if you have never used/seen such an if-then-else statement before, the above code can be read almost as an English sentence. Note how we indent things after if and after else (the else part can be omitted if we only want to do something if a condition is true) to group the code that we want to be executed in each case.

**Comment.** One potential issue with using the product  $f(a)f(c)$  rather than directly comparing the signs is that, depending on our available precision,  $f(a)f(c)$  might run into an underflow (note that  $f(a)$  and  $f(c)$  are each getting smaller by construction) and be treated as zero. Here is a simple artificial example illustrating the issue:

```
>>> def f(x):
    return (x-1)**99

>>> f(0.99) < 0

True

>>> f(1.01) > 0

True

>>> f(0.99) * f(1.01) < 0

False
```

With the representation of floats and their precision in mind, explain the above output!

When writing serious code, we therefore have to account for this and should instead compare the sign of  $f(a)$  to the sign of  $f(b)$  (and not compute the product). We will ignore this issue in the sequel.

**Example 34.** Python Let us implement the bisection method in Python (using the observation from the previous example). As input, we use a function  $f$ , the end points  $a$  and  $b$  of an interval guaranteed to contain a root of  $f$ , and the number of iterations that we wish to perform. The output is the final interval.

```
>>> def bisection(f, a, b, nr_steps):
    for i in range(nr_steps):
        c = (a + b) / 2
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return [a, b]
```

In order to use this function, we need to define a function  $f(x)$ . For comparison with our computation in Example 31, let us define  $f(x) = x^3 - 2$ . We can call it anything.

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

Let us verify (always check simple examples when writing code!) the definition of  $f(x)$  by computing the values for  $x = 1$  and  $x = 2$ .

```
>>> my_f(1)
-1
>>> my_f(2)
6
```

We are now ready to perform bisection with this function on the interval  $[a, b]$  with  $a = 1$  and  $b = 2$ .

```
>>> bisection(my_f, 1, 2, 1)
[1, 1.5]
>>> bisection(my_f, 1, 2, 2)
[1.25, 1.5]
```

This matches our computation in Example 31.

**Example 35.** `Python` Our computation in the previous example automatically ended up using floats (we started with integer values for  $a$  and  $b$  but we end up with floats after dividing by 2 for the midpoint). To work with fractions (no rounding!) in Python, we can use the `fractions` module as follows.

```
>>> from fractions import Fraction
>>> Fraction(1, 2) + Fraction(1, 3)
5/6
```

[You will probably see `Fraction(5, 6)` as the output rather than the above prettified version.]

Remarkably, our code can be used with fractions without changing it in any way:

```
>>> bisection(my_f, Fraction(1), Fraction(2), 1)
[Fraction(1, 1), Fraction(3, 2)]
>>> bisection(my_f, Fraction(1), Fraction(2), 2)
[Fraction(5, 4), Fraction(3, 2)]
```

Now, this perfectly matches our computation in Example 31.

**Advanced comment.** Unlike some other programming languages, Python does not make us specify the type of variables. For instance, we never had to tell Python whether the variables  $a$  and  $b$  should be an integer or a float or something else. Instead, Python is flexible (and we just used that to our advantage). This is called **duck typing** (true to the idiom that *if it walks like a duck and it quacks like a duck, then it must be a duck*). As such, Python allows the variables  $a$  and  $b$  to be any type for which our code (for instance,  $(a + b) / 2$ ) makes sense. [As always, these features have advantages and disadvantages. For instance, disadvantages of duck typing are speed and safety (as in making problematic kinds of bugs more likely).]

Finally, let us compute all 4 iterations of Example 31 at once:

```
>>> [bisection(my_f, Fraction(1), Fraction(2), n) for n in range(1,5)]
[[Fraction(1, 1), Fraction(3, 2)], [Fraction(5, 4), Fraction(3, 2)], [Fraction(5, 4),
Fraction(11, 8)], [Fraction(5, 4), Fraction(21, 16)]]
```

**Example 36.** `Python` Note that our bisection method code in Example 34 evaluates the function  $f$  twice per iteration (because we compute  $f(a)$  and  $f(c)$ ). Rewrite our code to only require one evaluation of  $f$  per iteration.

**Solution.**

```
>>> def bisection(f, a, b, nr_steps):
    fa = f(a)
    for i in range(nr_steps):
        c = (a + b) / 2
        fc = f(c)
        if fa*fc < 0:
            b = c
        else:
            a = c
            fa = fc
    return [a, b]
```

Try it! In terms of output, it should behave exactly as our previous code.

So why is this an improvement? (At first glance, it just looks more complicated...) Well, we need to keep in mind that the function  $f$  could potentially be very costly to evaluate ( $f$  doesn't have to be a simple function as it was in Example 31; instead, the definition of  $f$  might be a long computer program in itself and might require things like querying databases in a complicated way). In such a case, this small detail might make a huge difference.

## The regula falsi method

As before, we wish to find a root of the continuous function  $f(x)$  in the interval  $[a, b]$ .

The **regula falsi method** proceeds like the bisection method.

However, as an attempt to improve our approximations, instead of using the midpoint  $\frac{a+b}{2}$  of the interval  $[a, b]$ , it uses the root of the secant line of  $f(x)$  through  $(a, f(a))$  and  $(b, f(b))$ .

**Comment.** Note that the root of the secant line will be a good approximation of the root of  $f(x)$  if  $f(x)$  is nearly linear on the interval.

**Example 37.** Derive a formula for the root of the line through  $(a, f(a))$  and  $(b, f(b))$ .

**Solution.** The line has slope  $m = \frac{f(b) - f(a)}{b - a}$ .

Since it passes through  $(a, f(a))$ , the line has the equation  $y - f(a) = m(x - a)$ . (We could, of course, also pick the other point and the final result will be the same.)

To find its root (or  $x$ -intercept), we set  $y = 0$  and solve for  $x$ .

We find that the root is  $x = a - \frac{f(a)}{m} = a - \frac{(b-a)f(a)}{f(b)-f(a)} = \frac{af(b) - bf(a)}{f(b) - f(a)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$ .

**Comment.** Note that the final formula  $\frac{af(b) - bf(a)}{f(b) - f(a)}$  for the root is symmetric in  $a$  and  $b$ . (As it must be!)

**Historical comment.** Regula falsi (also called *false position method*) derives its somewhat strange name from its long history where the above formula (in a time where formulas in the modern sense were not yet a thing) was used to solve linear equations  $f(x) = Ax + B = 0$  (where  $A$  and  $B$  typically wouldn't be known explicitly) by choosing two convenient values  $x = a$  and  $x = b$  (these would be the *false positions* since they are not the root itself).

[https://en.wikipedia.org/wiki/Regula\\_falsi](https://en.wikipedia.org/wiki/Regula_falsi)

To cut each interval  $[a, b]$  into the two parts  $[a, c]$  and  $[c, b]$ :

- Bisection uses the midpoint  $c = \frac{a+b}{2}$ .
- Regula falsi uses  $c = \frac{af(b) - bf(a)}{f(b) - f(a)}$ .

**Example 38.** Determine an approximation for  $\sqrt[3]{2}$  by applying the regula falsi method to the function  $f(x) = x^3 - 2$  on the interval  $[1, 2]$ . Perform 3 steps.

**Solution.** Note that  $f(1) = -1 < 0$  while  $f(2) = 6 > 0$ . Hence,  $f(x)$  must indeed have a root in the interval  $[1, 2]$ .

- $[a, b] = [1, 2]$  contains a root of  $f(x)$  (since  $f(a) < 0$  and  $f(b) > 0$ ).  
The regula falsi point is  $c = \frac{af(b) - bf(a)}{f(b) - f(a)} = \frac{8}{7}$ . We compute  $f(c) = -\frac{174}{343} < 0$ .  
Hence,  $[c, b] = \left[\frac{8}{7}, 2\right]$  must contain a root of  $f(x)$ .
- $[a, b] = \left[\frac{8}{7}, 2\right]$  contains a root of  $f(x)$  (since  $f(a) < 0$  and  $f(b) > 0$ ).  
The regula falsi point is  $c = \frac{af(b) - bf(a)}{f(b) - f(a)} = \frac{75}{62}$ . We compute  $f(c) = -\frac{54781}{238,328} < 0$ .  
Hence,  $[c, b] = \left[\frac{75}{62}, 2\right]$  must contain a root of  $f(x)$ .
- $[a, b] = \left[\frac{75}{62}, 2\right]$  contains a root of  $f(x)$  (since  $f(a) < 0$  and  $f(b) > 0$ ).  
The regula falsi point is  $c = \frac{af(b) - bf(a)}{f(b) - f(a)} = \frac{37,538}{30,301}$ . We compute  $f(c) < 0$ .  
Hence,  $[c, b] = \left[\frac{37,538}{30,301}, 2\right]$  must contain a root of  $f(x)$ .

After 3 steps of the regula falsi method, we know that  $\sqrt[3]{2}$  must lie in  $\left[\frac{37,538}{30,301}, 2\right] \approx [1.2388, 2]$ .

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

**Example 39.** `Python` We can easily adjust our code from Example 34 for the bisection method to handle the regula falsi method. We only need to change the line that previously computed the midpoint  $c = \frac{a+b}{2}$  and replace it with  $c = \frac{af(b) - bf(a)}{f(b) - f(a)}$  instead:

```
>>> def regulafalsi(f, a, b, nr_steps):
    for i in range(nr_steps):
        c = (a*f(b) - b*f(a)) / (f(b) - f(a))
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return [a, b]
```

**Comment.** This code is using 6 function evaluations per iteration. As we did in the case of the bisection method, rewrite the code to only use a single function evaluation per iteration.

Let us use this code to automatically perform the computations we did in Example 38. Again, we use fractions to get exact values for easier comparison.

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

>>> from fractions import Fraction

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 1)
[Fraction(8, 7), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 2)
[Fraction(75, 62), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 3)
[Fraction(37538, 30301), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 4)
[Fraction(1534043307, 1226096954), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 5)
[Fraction(15236748520786296242, 12128315482217382469), Fraction(2, 1)]
```

No wonder that we stopped after 3 iterations by hand...

**Important comment.** The final two outputs give us an indication why performance-critical scientific computations are usually done using floats even if all involved quantities could be exactly represented as rational numbers. Compute the next iteration! The numerator and denominator integers can no longer be stored as 64 bit integers. And this after a measly 6 iterations of a simple algorithm!

This is a typical problem with any exact expressions. In practice, their complexity (the number of bits required to store them) often grows too fast.