question 2 views

Daily Challenge 14.6

(**Due: Wednesday 9/19 at 12:00 noon**)

Today we'll start implementing Newton's method in Python!

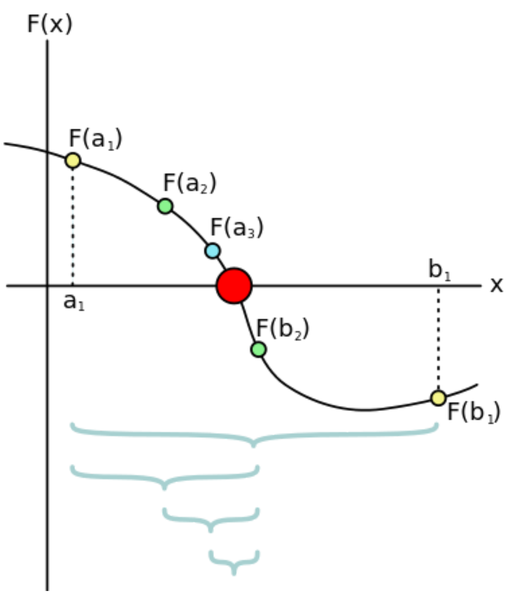
(1) Root-finding is important.

We saw in session 32 that every "solve this equation" problem -- that is, every question which can be phrased in the form "find a value of x such that $f(x) = g(x)$, where f and g are two expressions that I will hand you" -- can be rephrased as a root-finding problem, i.e. a question of the form "find a value of x such that $h(x) = 0$."

Indeed, perhaps this is obvious since we can always choose $h(x) = f(x) - g(x)$, which reduces the second problem to the first.

So if you believe that solving equations is important, you must necessarily believe that root-finding is *at least* as important, since equation-solving problems are a subset of root-finding problems.

The most naive approach to root-finding is bisection. The idea is simple: begin with an interval $[a_1, b_1]$ on which you know the function $h(x)$ has a root, typically because h is continuous and it changes sign on the interval. For clarity, in this discussion we will assume that $h(a_1)$ is positive and $h(b_1)$ is negative. By the IVT, there exists some $c \in (a_1, b_1)$ where $h(c) = 0$. We wish to find that root c to some desired accuracy, say 0.001.



We shall guess that the root is halfway between a_1 and b_1, and then compute the value $h\left(\frac{a_1+b_1}{2}\right)$. If this guess is still positive, we know that the root must actually lie in $\left[\frac{a_1+b_1}{2}, b_1\right]$, so we will define this as our next interval. If instead the guess is negative, we know that the true root lies in $\left[a_1, \frac{a_1+b_1}{2}\right]$, so we make this the new guess.

We continue in this way, cutting the interval in half, until eventually the function output at our midpoint is within the given tolerance (say, 0.001) of zero.

Of course, this is a very naive method, but I will ask you to implement it in Python as a benchmark against which to compare Newton's method, which also uses information about the derivative.

(2) Problem: bisection.

Write a Python function which implements bisection search.

Your function should take three inputs: an object which is itself another Python function, a left endpoint, and a right endpoint. The function signature should look like

```
def bisection(f, left, right):
        '''
        Implements bisection search on the function f between the endpoints left and right

        Inputs:
                f: a Python function which maps floats to floats
                left: the left endpoint of some interval on which the function f changes sign
```

```
            right: the right endpoint of some interval where f changes sign

        Returns:
            A number x such that f(x) is an approximate root (i.e. |f(x)|<0.001)

            Otherwise, throws an error if the function does not change sign on [left, right]
        '''

        ## Your awesome code goes here
```

For example, suppose I define the function "cube" as follows:

```
def cube(x):
        return x*x*x
```

If I run your function on "cube" -- that is, if I evaluate `bisection(cube, -1, 1)` -- then it should return the value 0, since this is the unique root of $f(x) = x^3$ on $[-1, 1]$.

As another example, suppose I define the function "sine" by

```
import numpy as np

def sine(x):
        return np.sin(x)
```

If I call `bisection(sine, 2.5, 3.5)`, I should get back something close to π, since π is the unique root of $\sin(x)$ on $[-2.5, 3.5]$.

As a non-example, if I call `bisection(cube, 1, 2)`, your code should throw an error because the function x^3 does not change sign on the interval $[1, 2]$.

Note that this method will not find certain roots with multiplicity, where a function just barely touches the axis but does not cross. If I define the function

```
def square(x):
        return x*x
```

and call `bisection(square, -1, 1)`, your code should throw an error because $(-1)^2 = 1 = (1)^2$, so the function does not change sign on this interval (even though there is a root).

Roughly speaking, you should iterate a process which continually chooses the midpoint of the interval under consideration, finds the value of f at that midpoint, and then cuts the interval in half appropriately. Write in some logic that returns an error if the function f does not change sign on the given interval.

Otherwise, keep iterating until you find an input at which the value of the function output is close to zero (by which I mean less than 0.001 in absolute value), and then return that input.

daily_challenge

Updated 6 months ago by Christian Ferko

the students' answer, *where students collectively construct a single answer*

Done, check github.

Updated 6 months ago by Logan Pachulski

the instructors' answer, *where instructors collectively construct a single answer*

My code follows.

```
import numpy as np

def bisection(f, a, b, tolerance=1e-4):
    '''
    Implements bisection search on the function f between the endpoints left and right

    Inputs:
        f: a Python function which maps floats to floats
        a: the left endpoint of some interval on which the function f changes sign
        b: the right endpoint of some interval where f changes sign

    Returns:
        A number x such that f(x) is an approximate root (i.e. |f(x)|<0.0001)

        Otherwise, returns false if the function does not change sign on [a, b]
    '''
```

```python
    if f(a) == 0:
        return a

    if f(b) == 0:
        return b

    if np.sign(f(a)) == np.sign(f(b)):
        print("The function does not change sign on this interval")
        return False

    this_a = a
    this_b = b
    this_x = (this_a + this_b)/2

    while np.abs(f(this_x)) > tolerance:

        this_x = (this_a + this_b)/2

        if np.sign(f(this_x)) == np.sign(f(this_a)):
            this_a = this_x

        if np.sign(f(this_x)) == np.sign(f(this_b)):
            this_b = this_x

    return this_x
```

Updated 6 months ago by Christian Ferko

followup discussions *for lingering questions and comments*