question 2 views

Daily Challenge 14.7

(**Due: Thursday 9/20 at 12:00 noon eastern**)

Now that you have implemented ~~bisexual~~ bisection search, let's compare to Newton's method.

(1) Newton's method uses the tangent line approximation to find roots.

The idea of Newton's method was presented in session 32: suppose you want to find a root of some function f (that is, a point x such that $f(x) = 0$). Begin with an initial guess (sometimes called a *seed*) x_0. Then write down the tangent line approximation to f near x_0, namely

$$f(x) \approx f(x_0) + f'(x_0) \cdot (x - x_0).$$

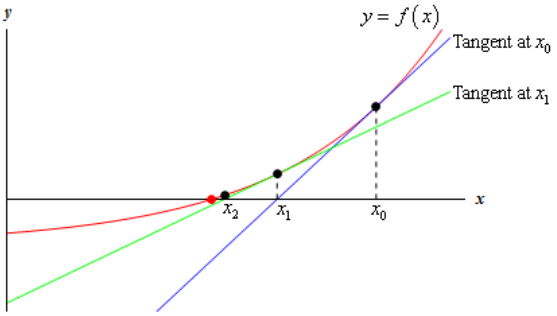
To approximate the root of f more accurately, we reason, perhaps we should find the root of the tangent line. Setting the above expression to zero gives

$$f(x_0) + f'(x_0) \cdot (x - x_0) = 0,$$

or, solving for x,

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Graphically, all that we've done is find the blue curve in the picture below and then solve for the point $x \equiv x_1$ which should be a "better" approximation to the root.



Of course, this process can be iterated. Now that we have a new guess x_1, why not find the tangent line approximation at x_1 and solve for its root to find a new guess x_2?

This is the idea behind *Newton's method*. More formally,

1. Begin with a seed x_0.
2. Then at each step x_n, compute $f(x_n)$ and $f'(x_n)$, then define the next point by

   $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

3. Continue until your current guess is close enough to a root, given your desired accuracy. For instance, if your tolerance is 0.001, stop when $|f(x_n)| < 0.001$.

I believe that the only way to really understand an algorithm is to implement it, so let's do that.

(2) Problem: implementing Newton's method.

Code up Newton's method in Python.

There are several design decisions to make here, which will require some creativity on your part. I will not tell you which decisions I think you should make, but I will make a few comments about the difficulties and offer some suggestions.

First, I suggest making the function signature similar to your implementation of bisection search to facilitate easy comparison. That is, your function might look like

```
def newton(f, left, right):
        '''
        Implements Newton's method to find a root of 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 has a root
        right: the right endpoint of some interval where f has a root

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

        Otherwise, throws an error if it cannot find a root
    '''

    ## Your awesome code goes here
```

The first problem you'll face is: how do I choose my initial guess (or seed) x_0? The most naive thing to do is simply take the average of the left and right endpoints, as in bisection search, but there are other options.

The next, and more interesting, question: how do I approximate the derivative $f'(x_0)$? Again, the most naive thing to do is to pick some small number h and then compute $\frac{f(x_0+h)-f(x_0)}{h}$. If h is too small, you will get numerical precision errors.

There are many smarter methods besides the "pick some small h and just compute" method. Some of them are discussed in the Wikipedia article on numerical differentiation. The five-point stencil is especially nice, which I alluded to in session 32.

Another option for finding the derivative is to use a built-in function like numpy's gradient function, but you'll need to read the documentation and figure out how it works. It's not as simple as just passing your function and a point as arguments into gradient.

Once you've solved the above problems, it should be pretty simple to implement the formula $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. But you should put in some logic to handle a few edge cases. For instance, what happens if that formula gives you a new point x_{n+1} which lies outside the interval $[\text{left}, \text{right}]$? In that case, you know Newton's method isn't giving you a better estimate.

Another issue is getting stuck in loops. If your function has gone through 1000 iterations and still hasn't found the root, something is probably wrong -- perhaps you put in some logic to try again with a different initial guess.

In short, there are many clever things you can do to improve the algorithm. Try as many as you have the patience for, and when you're happy with the result, test it on a few functions and see whether it gets the correct root (and how quickly, i.e. how many iterations it takes).

daily_challenge

Updated 6 months ago by Christian Ferko

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

My code follows. Check out the Jupyter notebook (.ipynb here, .pdf here) where I test it.

```
import numpy as np

epsilon = 2.220446049250313e-16
## This is my tested value of the machine epsilon

def differentiate(f, x):
    """
    Returns an approximate first derivative f'(x) at the point x

    Inputs:
        f: a Python function which maps floats to floats
        x: a point in the domain of f

    Returns:
        An approximation to f'(x) using the five-point stencil

    """
    h = np.sqrt(epsilon*max(x, 0.1)) ## Don't want h=0 if x=0

    stencil = ( -f(x+2*h) + 8*f(x+h)-8*f(x-h) + f(x-2*h) )/(12*h)

    return stencil

def newton(f, a, b, tolerance=1e-4):
    '''
    Implements Newton's method to find a root of 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 has a root
        b: the right endpoint of some interval where f has a root

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

        Otherwise, returns false
    '''

    ## Rather than guessing halfway between a and b, try a few values and pick the smallest
```

```
    test_inputs = np.linspace(a, b, 100)
    test_outputs = f(test_inputs)

    smallest_index = np.argmin(np.abs(test_outputs))

    ## Start with the smallest output, should be closest to the root

    xn = test_inputs[smallest_index]

    iters = 0

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

        xn = xn - (f(xn))/(differentiate(f, xn))

        iters += 1

        if iters > 1000 or xn > b or xn < a:
            xn = np.random.choice(test_inputs)

            ## If something goes wrong, try a different seed

        if iters > 10000:

            ## If it doesn't work after 10,000 tries, quit

            print("Could not find a root")

            return False

    return xn
```

Updated 6 months ago by Christian Ferko

followup discussions *for lingering questions and comments*