CS98-52 Lecture 1

Wednesday, September 4th, 2019

Deadline to add: September 18th, 2019

website: cs61a.org/extra.html

Newton's Method

Background

Using higher-order functions in Newton's method.

- Quickly finds accurate approximations to zeroes of differentiable functions.
-
- There exists two zeroes of this graph, $\sqrt{\ }$
- A method for computing square roots, cube roots, etc.
- The positive zero is is $\sqrt{\ }$. (We're solving the equation).

Newton's Method

- Given a function `f` and initial guess `x`, we improve our guess every time:
  - Compute the value of at the guess: f(x)
  - Compute the derivative of f at the guess: f'(x)
  - Update guess x to be: —
  - Finish when f(x) is close enough
- This method works by pretending the line of the derivative is a curve, finding the x-point of the zero of the derivative, finding the y-point of the curve at that x, and repeat.
- Failure case when the guess is at a tangent line.
- Very effective to find precise answers, vs bisection search

Using Newton's Method

- How to find the square root of 2?

```
>>> f = lambda x: x*x - 2
>>> df = lambda x: 2*x
>>> find_zero(f, df)
1.41421356...
```

- Find the cube root of 729

```
>>> g = lambda x: x*x*x - 729
>>> dg = lambda x: 3x*x
>>> find_zero(g, dg)
9.0
```

Beforew Newton, the Babylonian Method iterative refined a guess x about the square root of a.

Let's build this function.

```
square = lambda x: x * x
def square_root(a):
    x = 1
    while square(x) != a:
        x= square_root_update(x,a)
    return x

def square_root_update(x,a):
    return (x+a/x) / 2
```

If we put it `square_root(16)` , and put a print statement, we can see the computer guessed 1, 8.5, 5.2..., and then converges to 4.

But this function doesn't do well with non-perfect squares. `square_root(2)` puts it into an infinite loop.

Let's move on to cube roots first.

```
def cube_root(a):
    x+1
    while x**3 != a:
        x = cube_root_update(x,a)
    return x

def cube_root_update(x, a):
    return (2*x+a/(x*x))/3
```

We see a lot of repetition right now between the square and cube functions, so we could probably generalize.

```
def improve(update, close, guess=1):
    while not close(guess):
        guess = update(guess)
    return guess
```

How do we solve our infinite loop problem? We can do this with a tolerance (to a certain decimal place, if close enough, it ends).

```
def approx_eq(x, y, tolerance=1e-15):
    return abs(x-y) < tolerance
```

We can now write our new square function.

```
def square_root(a):
    def update(x):
        return square_root_update(x,a)
    def close(x):
        return approx_eq(x*x, a)
    return improve(update, close)
```

And our new cube function too:

```
def cube_root(a):
    return improve(lambda x: cube_root_update(x,a), lambda x: approx_eq(x**3, a))
```

Let's try to implement Newton's method, and try to get rid of the Babylonian formulas.

```
def find_zero(f,df):
    def near_zero(x):
        return approx_eq(f(x),0) #A zero of x is a place where f(x) is close to 0
    return improve (newton_update(f,df), near)zero)
```

And the newton_update is a generic replacement for the previous formulas.

```
def newton_update(f, df):
    return lambda x: x - f(x)/df(x)
```

Why is it a function? It's so that `improve` above can have a function to call.

We can now update `square_root` function:

```
def square_root(a):
    def f(x):
        return x * x - a
    def df(x):
        return 2 * x
    return find_zero(f,df)
```

Approximate Differentiation

We can get rid of the manual calculation needed to derive a function by writing code to do it automatically.

```
def slope(f, x, a=1e-10):
    return (f(x+a)-f(x))?a

def derive(f):
    return lambda x: slope(f, x)
```

Now, `find_zero` doesn't need `df` , we just need to compute it.